DEPARTMENT OF STATISTICS

LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Master's Thesis in Statistics

Band Matrices in Recurrent Neural Networks for Long Memory Tasks

Johannes Langer

DEPARTMENT OF STATISTICS

LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Master's Thesis in Statistics

Band Matrices in Recurrent Neural Networks for Long Memory Tasks

Band Matrizen in Rekurrenten Neuronalen Netzwerken für Langzeitgedächtnis Aufgaben

Author:	Johannes Langer
Supervisor:	Bernd Bischl
Advisor:	Sebastian Urban
Submission Date:	11. October 2017

I confirm that this master's thesis in statistics is my own work and I have documented all sources and material used.

Munich, 11. October 2017 Johannes Langer

Acknowledgments

A very special gratitude goes out to my advisor Sebastian Urban. Working with him was an absolute inspiration. Special thank goes to my supervisor Bernd Bischl for offering me the chance to work on this thesis and his support despite of being fully booked.

Abstract

Recurrent neural networks face difficulties to develop a proper long-term memory structure. End-to-end sequence learning tasks with distorted input information and (consequentially) unaligned labels are especially hard to learn. We argue that tasks of this type require the model to have a strong capacity to store input history, which correlates with the number of hidden units.
The braille data set, offers a good testing environment of this type, consisting of long sequential data of tactile sensors capturing braille characters. It also provided the possibility to increase its difficulty by artificial and natural distortions.
For long-memory tasks we propose a model with classical RNN architecture, but with the restriction of a recurrent band matrix (band RNN). We assume that models of this type offer all necessary features to perform well in the given testing environment: offering tractability up to high numbers of hidden units, increased input history capacity and damped growth in task-storage capacity and overall complexity.
We further extend the model by slightly loosening up the band matrix restriction. Through the combination with a latticed-like sparse matrix, the model inherits global dependencies between the hidden units (combination RNN). Otherwise we combined the band matrix with small upper respectively lower triangular matrices in the out-of-band corners (closed RNN). This builds a circular dependency of hidden units and the possibility to use shift initialization. Both extensions keep their tractability for the case of a very large number of hidden units.
To fully utilize this tractability we implemented two efficient CUDA kernels for the required operations, namely band-matrix matrix multiplication SGBMM and the matrix to band-matrix multiplication SB2BMM. Our kernels achieved cuts in computation time over 95% compared to CUBLAS SGEMM for matrix sizes of 6144. Moreover we successfully included them as new operations into the *F#* deep learning framework DeepNet.
Our experiments confirmed the expectation that the braille task requires increased input history storage capacity. Delayed prediction of the task specific output was required to achieve good performance. Finally we found that the closed RNN with shift initialization offers the necessary qualities to develop long-term memory structure, achieving the highest accuracy in the braille task.

Contents

1. Introduction

1.1. Motivation

In 1982 the futurist John Naisbitt asserted: *"We are drowning in information, but starved for knowledge."* [Nai82]. Over three decades later, the increase in computational performance and storage kept up the exponential paste predicted in Moore's "law". The wide spread of computational devices and the strongly developed communication systems lead to huge amounts of data, collected in various fields of application. Recently [Bro14] argued: *"We are drowning in data, but starved for information."* and that we demand algorithms suitable to process the massive amounts and very differing types of data. Consequently the field of data analysis saw an explosive growth of machine learning algorithms, which shift the focus away from statistically sound theories to applicability and computability. A collection of very successful machine learning algorithms unites under the name of neural networks or deep learning. Convolutional neural networks [LL98], tailored to work with spatial data (common in computer vision), achieved excellent performance in image recognition [Sze14] [HX15] and segmentation [RS16]. Recurrent neural networks (RNNs) and especially long short-term memory networks (LSTMs) [HS97], designed to work with sequential data, were widely successful in speech recognition [BK15] and machine translation [AR15]. Nevertheless, RNNs face difficulties when in comes to very long sequences, which require the model to develop a capable memory structure. The objective of this thesis was to improve the ability of RNNs to develop a long-term memory mechanism.

1.1.1. Sequence Learning

Sequential learning tasks like speech recognition and translation are difficult problems for machines. Especially hard is the case of end-to-end sequence learning in which input and labels are sequential. Models are required to develop a memory mechanism in its states and parameters, capable to store and read out information. We can draw an analogy between algorithmic and human memory, to explain the difficulty lying in that task. With a strong memory we could remember every last detail, but would likely fail to identify profound patterns, obvious for others to perceive. With a holey memory on the other hand we may forget crucial details, essential to make an accurate decision (prediction). Depending on the specific task, a balance between learning new and forgetting old information has to be found. If the data generating process of the task can hold their assumptions, probabilistic methods like Hidden Markov Models [EP66] (current state depends only on previous state) or Gaussian Processes are

reasonable choices to model the sequence. Neural networks include the extraction of features about the data generating process into the training, allowing to spare further assumptions. They are proven to inherit the property to work as universal approximator [Cyb89] [HM89]. Moreover, for RNNs can be shown that they can be Turing complete [TD95], capable to execute arbitrary computations. However, research has shown that the training of its parameters, determined by its mathematical structure, is prone to ignore information lying further in the past [BSF94]. Incapable of learning to take past information into account they are stuck with overfitting on the current information.

1.1.2. Unaligned Labels

The development of long-term memory becomes more important for tasks in which the labels are unaligned to the sequential inputs. This can happen in a shifted (Copying task [HS97]) or a completely distorted way (Braille task 7.4), illustrated in Figure 1.1. In both cases a delay is needed to guarantee that the information required to predict the label was already received. Moreover, in the distorted case the model needs to develop a memory structure in which it aggregates the information in an ordered manner. Without the knowledge where to predict the labels (at which step in the sequence), we have to assume their dependency of the whole input sequence. The delay becomes a separate hyper-parameter, characterizing the trade-off between *"having received all necessary information"* and *"still having it in memory when it is required"*. The difficulty of this task increases with the sequence length and the heaviness of the distortions. Tasks with unaligned labels come in handy for our objective to find RNN extensions with strong long term memory capabilities. Note that we dropped approaches which try to realign the labels in an extra step (e.g. connectionist temporal classification [GS06] or hidden markov models), since this would decrease the pressure to form long term memory.

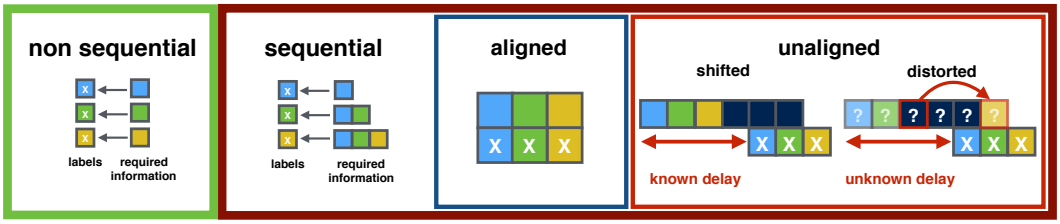

Figure 1.1.: Sequential data and unaligned labels.

1.2. Development of long term memory structures in RNNs

RNNs, including gated architectures, store information about the task in their parameters and information about the input history in their units/state dimensions. The work of

[CJ17] found that for all RNN models the amount of stored task information and input history grows linearly in the number of parameters (5 bits per parameter) respectively in the number of hidden units (1 real value per unit). Moreover they argue that primarily the task-storage capacity determines the model performance. Since all models have the same capacity behavior, they conclude that the models' difference in success lies in trainability. Identified as being the principal cause of a limited trainability is the vanishing and exploding gradient problem [BSF94]. Its solution was a main driving force in the development of new models. Divergent to this trend, this thesis primarily focuses on the development of a long-term memory structure. We are therefore especially interested in tasks for which the storage capability of input history becomes increasingly important. Without directly solving the vanishing and exploding gradient problem, we try to find a model which is robustly trainable with a big storage capacity of input history. We suggest that a classical RNN architecture using a band-matrix as recurrent weight matrix, could be such a model.

1.2.1. Vanishing and Exploding Gradient Problem

Although our model is incapable of directly solving the vanishing and exploding gradient problem, we have to understand how strong it will impact our model. We will give a detailed discussion about the problem in 3.2.2 and continue to present some proposed solutions. Approaches to minimize the effect of exploding gradients include gradient clipping and the use of L_1 or L_2 weight norm penalty terms [PT13]. Instead of penalizing the weights directly, penalization of differences in successive norm pairs in the forward (activations) [KM15] or [PT13] backward (derivatives) pass has been proposed. Orthogonal (ORNN) [HA16] and identity (IRNN) [LN15] initialization were investigated as well as the restriction of a unitary recurrent weight matrix (URNN) [AA16] [WT16] [VC17]. Widely spread are gated architectures like long-short term memory networks (LSTM) [HS97] or gated recurrent units (GRU) [CB14], which alter the classical architecture of RNNs to phase out vanishing and exploding gradients.

1.2.2. Problems concerning Large Hidden States

The hidden state correlates with the models capacity to store input history [CJ17]. Due to our focus on long-term memory, our tasks require a comparable higher capacity to store previous input information. Consequentially we have to increase the number of hidden units. Classical models suffer of two problems when facing large hidden states of size m:

- The models become intractable due to the growth in computational cost. In 2.4 we will derive that the storage and the number of operations required by classical models grows with $O(m^2)$ (squared growth in the limit of m). Figures 7.2.1 and 7.2.2 show speed benchmarks for matrix multiplication respectively model training.
- The models become prone to overfitting due to their growth in complexity. The

number of parameters is linearly correlated with the model capacity to store task information [CJ17]. In Figure 7.1 we show the trend of the overall number of parameters depending on the number of hidden units for fixed other model hyper-parameters.

Adding regularization terms like $L1$ or $L2$ penalty (decreasing complexity through the effect of bounding the spectral radius of the linear transformations), as well as dropout [ZI15], would increase the computational costs even further. Factorization tricks for RNNs [LV16] and LSTMs [KG17] successfully reduce both effects. Although, if the compression of the high dimensional hidden state into a much lower factoring dimension didn't affect the models' performance, it maybe didn't require a larger hidden state in the first place.

1.3. Thesis Flow

The mentioned problems are rooted in mathematics, determined by methodical architectures and their solution demanded an efficient technical implementation. To enable a profound discussion we give prerequisites to each area in 2, 3 and 4. We continue to illustrate our idea of band RNNs and possible extensions in 5. A large share of work time was spent on an efficient implementation of a band-matrix matrix multiplication kernel, described in 6. We then present our experimental results in 7 and finish this thesis with a discussion in 8.

2. Mathematical Prerequisites

The restriction to a recurrent weight band matrix changes the mathematical and computational properties of the model. On the lowest level we have to explore the change in matrix properties 2.4, requiring some basics in linear algebra 2.1. The integration of our band-matrix matrix multiplication kernel into a deep learning framework demands us to think about the gradient of this operation 2.3. The necessary basics of differential calculus are presented in 2.2. All mathematical definitions can be found in slightly varying form in the Handbook of Mathematics [Ily45].

2.1. Linear Algebra

Definition 2.1 (Linear Map)**.** *Let V and W be vector spaces over the body $\mathbb{K}$. A transformation $T : V \to W$ is called $\mathbb{K}$-linear, if for all $u, v \in V$ and $\lambda \in \mathbb{K}$ holds:*

$$T(u + \lambda v) = T(u) + \lambda T(v) \tag{2.1}$$

The set of all linear maps from V to W is denoted $\mathcal{L}(V, W)$.

Definition 2.2 (Matrix)**.** *Let m and n denote positive integers. An m-by-n matrix A is a rectangular array of elements of $\mathbb{K}$ with m rows and n columns:*

$$A = \begin{pmatrix} A_{1,1} & \cdots & A_{1,n} \\ \vdots & & \vdots \\ A_{m,1} & \cdots & A_{m,n} \end{pmatrix} \tag{2.2}$$

The notation $A_{j,k}$ denotes the entry in row j, column k of A.

Definition 2.3 (Matrix of a linear Map)**.** *Suppose $T \in \mathcal{L}(V, W)$ and $v_1, ..., v_n$ is a basis of V and $w_1, ..., w_m$ is a basis of W. The **matrix of T** with respect to theses bases is the m-by-n matrix $\mathcal{M}(T)$ whose entries $A_{j,k}$ are defined by:*

$$T(v_k) = A_{1,k}w_1 + \cdots + A_{m,k}w_m \tag{2.3}$$

Definition 2.4 (Transpose Matrix)**.** *Suppose A is an m-by-n matrix. Then the transpose matrix A^T is an n-by-m matrix obtained from A by interchanging the rows and columns:*

$$A^T_{k,j} = A_{j,k} \tag{2.4}$$

Definition 2.5 (Matrix Multiplication)**.** *Suppose A is an m-by-r matrix and C is an r-by-n matrix. Then AC is defined to be the m-by-n matrix whose entry in row j, column k, is given by the following equation:*

$$(AC)_{j,k} = \sum_{i=1}^{r} A_{j,i} C_{i,k} \tag{2.5}$$

Definition 2.6 (Rank)**.** *Suppose A is an m-by-n matrix with entries in $\mathbb{K}$. The row rank of A is the dimension of the span of the rows of A in $\mathbb{K}^{1,n}$. The column rank of A is the dimension of the span of the columns of A in $\mathbb{K}^{m,1}$. W.l.o.g. we refer to the column rank of A just as rank of A, noted as* $\operatorname{rank}(A)$.

Remark. *Given its definition it is self-evident that the rank of A is a non-negative integer and cannot be greater than either m or n:*

$$0 \leq \operatorname{rank}(A) \leq \min(m, n) \tag{2.6}$$

Definition 2.7 (Full Rank)**.** *An m-by-n matrix A is said to have full rank if:*

$$\operatorname{rank}(A) = \min(m, n) \tag{2.7}$$

otherwise, the matrix is rank deficient.

Theorem 2.8 (Rank of Matrix Multiplication)**.** *Suppose A is an m-by-r matrix and B an r-by-n matrix, then*

$$\operatorname{rank}(AB) \leq \min(\operatorname{rank}(A), \operatorname{rank}(B)) \leq \min(m, r, n) \tag{2.8}$$

Definition 2.9 (Determinant)**.** *Suppose A is an n-by-n matrix. Then the determinant of A, denoted $|A|$, is defined by:*

$$|A| = \sum_{(m_1,\ldots,m_n)\in perm\, n} (sign(m_1, \ldots, m_n)) A_{m_1,1} \cdots A_{m_n,n} \tag{2.9}$$

Theorem 2.10 (Linear Maps change volume by factor of determinant)**.** *Suppose the linear map $T \in \mathcal{L}(\mathbb{R}^n, \mathbb{R}^n)$ and a volume of space $\Omega \subset \mathbb{R}^n$. Then:*

$$volume\, T(\Omega) = |T|(volume\, \Omega) \tag{2.10}$$

Definition 2.11 (Singular Value Decomposition)**.** *Suppose A is an m-by-n matrix over the body $\mathbb{K}$ ($\mathbb{R}$ or $\mathbb{C}$). Then there exists a factorization, called singular value decomposition of A, of the form:*

$$A = U\Sigma V^* \tag{2.11}$$

- *U is an m-by-m unitary matrix (if $\mathbb{K} = \mathbb{R}$, unitary matrices are orthogonal matrices).*
- *Σ is a diagonal m-by-n matrix with non-negative real numbers on the diagonal.*
- *V^* is the conjugate transpose of an n-by-n unitary matrix V.*

Theorem 2.12 (Eigenvalues and Determinant)**.** *Given the eigenvalues λ_i of matrix A, the determinant can also be derived as:*

$$|A| = \prod_{i=1}^{n} \lambda_i \tag{2.12}$$

Theorem 2.13 (Full Rank Equivalences)**.** *For an m-by-m square matrix A the following properties are equivalent:*

(a) A has full rank m.

(b) A is invertible.

(c) A is a bijective linear map (isomorphism).

(d) A has a non-zero determinant.

(e) A has no zero valued eigenvalues.

2.2. Calculus

Definition 2.14 (Partial Derivative)**.** *Suppose $U \subseteq \mathbb{R}^n$ is open and $f : U \to \mathbb{R}$ a scalar valued function. Then the partial derivative of f at point $\mathbf{a} \in U$ with respect to the i-th variable x_i is defined as:*

$$\frac{\partial f}{\partial x_i}(\mathbf{a}) = \lim_{h \to 0} \frac{f(a_1, ..., a_i + h, ..., a_n) - f(a_1, ..., a_i, ..., a_n)}{h} \tag{2.13}$$

If all partial derivatives exist in a neighborhood of $\mathbf{a}$ and are continuous there, then f is totally differentiable in that neighborhood. The vector of partial derivatives is called gradient:

$$\nabla f(\mathbf{a}) = (\frac{\partial f}{\partial x_1}, ..., \frac{\partial f}{\partial x_n}) \tag{2.14}$$

Definition 2.15 (Total Derivative)**.** *Suppose $U \subseteq \mathbb{R}^n$ is open and $\mathbf{f} : U \to \mathbb{R}^m$. Then the total derivative of $\mathbf{f}$ at point $\mathbf{a} \in U$ is the unique linear transformation $\mathbf{f}'(\mathbf{a}) : \mathbb{R}^n \to \mathbb{R}^m$ such that:*

$$\lim_{\mathbf{h} \to 0} \frac{||\mathbf{f}(\mathbf{a} + \mathbf{h}) - \mathbf{f}(\mathbf{a}) - \mathbf{f}'(\mathbf{a})\mathbf{h}||}{||\mathbf{h}||} = 0 \tag{2.15}$$

If the total derivative exists at $\mathbf{a}$, then all partial derivatives of f exist at $\mathbf{a}$. Using the coordinate functions $\mathbf{f} = (f_1, ..., f_m)$ the total derivative can be expressed as matrix, called the Jacobian matrix of $\mathbf{f}$ at $\mathbf{a}$:

$$\mathbf{f}'(\mathbf{a}) = \mathbf{Jac}_{\mathbf{f}}(\mathbf{a}) = (\frac{\partial f_i}{\partial x_j})_{ij} \tag{2.16}$$

Definition 2.16 (Chain Rule)**.** *The chain rule is a formula for computing the derivative of the composition of two or more functions. That is, if $g : X \to Y$ and $f : Y \to Z$ are differentiable functions, then the chain rule expresses the derivative of their composition $f \circ g = h$ with $h : X \to Z$ and $z = h(x) = f(y) = f(g(x))$:*

$$(f \circ g)' = (f' \circ g) \cdot g' \quad \Leftrightarrow \quad f(g(x))' = f'(g(x))g'(x) \quad \Leftrightarrow \quad \frac{\mathrm{d}z}{\mathrm{d}x} = \frac{\mathrm{d}z}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}x} \tag{2.17}$$

Definition 2.17 (Sum convention)**.** *Notational convention which determines that multiple usage of the same index requires the summation over this index. This allows to omit the summation symbol whenever an index is present more then once. E.g. matrix multiplication we can write:*

$$C_{ij} = \sum_{k=0}^{K} A_{ik}B_{kj} \overset{S.C.}{=} A_{ik}B_{kj}$$

Definition 2.18 (Chain Rule in Higher Dimensions)**.** *Suppose $g : X \to Y$ and $f : Y \to Z$ are differentiable functions between multivariate spaces $X \subseteq \mathbb{R}^n, Y \subseteq \mathbb{R}^m, Z \subseteq \mathbb{R}^k$ and $\mathbf{a} \in X$. g and f can now be expressed in terms of their components as $\mathbf{y} = g(\mathbf{x}) = (g_1(\mathbf{x}), ..., g_m(\mathbf{x}))$ and $\mathbf{z} = f(\mathbf{y}) = (f_1(\mathbf{y}), ..., f_k(\mathbf{y}))$. To derive the total derivative 2.15 of the composition $f \circ g = h$ with $h : X \to Z$ at $\mathbf{a}$ we generalize the chain rule:*

$$\mathbf{h}'(\mathbf{a}) = \mathbf{f}'(\mathbf{g}(\mathbf{a}))\mathbf{g}'(\mathbf{a}) \quad \Leftrightarrow \quad \mathbf{Jac_h}(\mathbf{a}) = \mathbf{Jac_f}(\mathbf{g}(\mathbf{a}))\mathbf{Jac_g}(\mathbf{a}) \tag{2.18}$$

Remark: Given a one dimensional loss function $h = \mathcal{L}$ which maps from the parameter-space $X = \Theta \subseteq \mathbb{R}^n$ to the loss space $\mathbb{R}$. In this case the Jacobian matrix of h at a point $\boldsymbol{\theta}^0$ is exactly the gradient of h respective to all parameters $(\theta_1, ..., \theta_n) \in \Theta$ at $\boldsymbol{\theta}^0$. With all other properties holding as before we can write:

$$\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}^0) = \mathbf{Jac}_f(\mathbf{g}(\boldsymbol{\theta}^0))\mathbf{Jac_g}(\boldsymbol{\theta}^0) =$$
$$(\frac{\partial \mathcal{L}(\mathbf{g}(\boldsymbol{\theta}^0))}{\partial y_1} ... \frac{\partial \mathcal{L}(\mathbf{g}(\boldsymbol{\theta}^0))}{\partial y_m}) \begin{pmatrix} \frac{\partial g_1(\boldsymbol{\theta}^0)}{\partial \theta_1} & \cdots & \frac{\partial g_1(\boldsymbol{\theta}^0)}{\partial \theta_n} \\ \vdots & & \vdots \\ \frac{\partial g_m(\boldsymbol{\theta}^0)}{\partial \theta_1} & \cdots & \frac{\partial g_m(\boldsymbol{\theta}^0)}{\partial \theta_1} \end{pmatrix} \overset{S.C.}{=} \frac{\partial \mathcal{L}(\mathbf{g}(\boldsymbol{\theta}^0))}{\partial y_i} \frac{\partial g_i(\boldsymbol{\theta}^0)}{\partial \theta_j} \tag{2.19}$$

2.3. Gradient Based Optimization

Deep Learning frameworks offer an abstraction to define functional graphs and the option to efficiently calculate the gradient respective to all variables. The surprisingly low cost of computation to derive all partial derivatives and the idea to base the learning of neural networks on it was first proposed by [RG86] and named backpropagation. Until today it is the standard way to optimize the parameters used in deep learning algorithms. Nevertheless a recent publication showed, that evolutionary strategies can compete with backpropagation when it comes to reinforcement learning [**evol**]. A more elaborate introduction of gradient based optimization can be found in [Mur12].

2.3.1. Backpropagation

To minimize confusion we want to clearly separate the terms forward pass and loss as well as backward pass and backpropagation. The term **pass** emphasizes the step-wise execution of the functional (computational) graph, which passes intermediate results from one node to the next. Although **loss** refers to the final node of the forward pass,

it emphasizes the measure used for optimization. Likewise the term **backpropagation** not only comprises the concept of efficiently deriving all partial gradients during the execution of a backward pass, but emphasizes its role during the training of neural networks.

Forward and Backward Pass

The Forward Pass is a function from input, output and parameter-space into loss-space:

$$\mathcal{FP} : X \,\mathrm{x}\, Y \mathrm{x}\, \Theta \rightarrow L \tag{2.20}$$

The Backward Pass is a function from input, output, parameter and loss-space to parameter-space:

$$\mathcal{BP} : X \,\mathrm{x}\, Y \mathrm{x}\, \Theta \mathrm{x}\, L \rightarrow \Theta \tag{2.21}$$

The Forward Pass includes the calculation of the model **prediction** as an intermediate result and the model **loss** as the final result. Subsequently the **loss** is taken as input of the backward pass. The complete flow through both passes, referred to as one backpropagation **step**, provides the gradient to update all model parameters. How the final parameter update is calculated based on this gradient depends on the used optimization method 2.3.3.

Efficiently using the Chain Rule

Backpropagation describes an efficient way to calculate all partial derivatives (the gradient) of a computational graph. Having the chain rule in mind we notice that the partial derivative of the final node is part of the product to calculate earlier nodes. We therefore start at the tail of the computational graph and propagate (store and reuse) the resulting derivatives back to calculate the others. Although being computationally more expensive then the forward pass, the backward pass achieves at least comparable performance. When we arrive at the input nodes we aggregated the partial derivatives of all parameters in the gradient vector, which is then taken as input by the gradient based optimization methods. All nodes are therefore required to contain differentiable functions and their derivatives have to be implemented. Suppose Backpropagation arrives at a certain node (operation) involving two variables $\mathbf{g}(\mathbf{x^{(1)}}, \mathbf{x^{(2)}}) = \mathbf{y}$, with $\mathbf{x^{(1)}}$ and $\mathbf{x^{(2)}}$ being in general results of earlier nodes. Further we describe all upcoming nodes (operations) with a function $f(\mathbf{y})$ so we can write our model loss the following way: $\mathcal{L}(\mathbf{x^{(1)}}, \mathbf{x^{(2)}}) = f(\mathbf{g}(\mathbf{x^{(1)}}, \mathbf{x^{(2)}}))$. The fact that backpropagation arrived at this node means that all partial derivatives of latter nodes are already derived, in our case $\frac{\partial \mathcal{L}(\mathbf{y})}{\partial y_k}$ $\forall k \in \{1, ..., m\}$. To further pass back the derivatives until the initial nodes are

reached, we need to calculate the derivatives with respect to $x^{(1)}$ and $x^{(2)}$:

$$\begin{aligned}\frac{\partial \mathcal{L}(\mathbf{x^{(1)}}, \mathbf{x^{(2)}})}{\partial x_i^{(1)}} &\overset{S.C.}{=} \frac{\partial \mathcal{L}(\mathbf{y})}{\partial y_k} \frac{\partial y_k}{\partial x_i^{(1)}} \\ \frac{\partial \mathcal{L}(\mathbf{x^{(1)}}, \mathbf{x^{(2)}})}{\partial x_i^{(2)}} &\overset{S.C.}{=} \frac{\partial \mathcal{L}(\mathbf{y})}{\partial y_k} \frac{\partial y_k}{\partial x_i^{(2)}}\end{aligned} \tag{2.22}$$

If we are interested in the case that $\mathbf{x^{(1)}} = A$ refers to an m-by-r matrix, $\mathbf{x^{(2)}} = B$ to an r-by-n matrix and the operation $g(A, B) = AB$ to matrix multiplication, we can write the derivatives as:

$$\begin{aligned}\frac{\partial \mathcal{L}(C)}{\partial A_{ij}} &\overset{S.C.}{=} \frac{\partial \mathcal{L}(C)}{\partial C_{ab}} \frac{\partial C_{ab}}{\partial A_{ij}} \\ \frac{\partial \mathcal{L}(C)}{\partial B_{ij}} &\overset{S.C.}{=} \frac{\partial \mathcal{L}(C)}{\partial C_{ab}} \frac{\partial C_{ab}}{\partial B_{ij}}\end{aligned} \tag{2.23}$$

W.l.o.g we thereby changed the indices from i and k to proper matrix indices. We could easily convert them back into a flattened version using $i + j * m$ for A, $i + j * r$ for B and $a + b * m$ for C. Note that this would correspond to the column-major order.

2.3.2. Gradient Descent

The theory of gradient based optimization originates from the idea of gradient descent. The goal is to optimize an objective function, in our case to minimize the **loss** of a deep learning model (compared to maximize the utility in other contexts). Different parameter values result in a different loss. We want to find the point in parameter space which results in the lowest loss value. Since we have no information where in parameter-space this position lies, we have to start searching at a random initialization. The gradient of the loss with respect to all parameters has the following important characteristics:

- The negated gradient points into the direction of parametrizations leading to the strongest decrease in loss value.
- The value of the gradient reaches zero at a local minimum and continuously shrinks as we approach it.
- Since the gradient vanishes at every local minima, following the gradient doesn't have to lead us to the global minima (if existing).

Starting from a random initialization and moving all parameters slightly into the direction of the negative gradient will decrease the loss. Repeating this step multiple times we should reach a local or, if existing, the global minimum. This leaves us with two problems:

1. How slightly should we move in the negative gradient direction? With a big step size we might leap the minimum every time we come closer. With a short step size we might never approach it in a reasonable amount of time.

2. Even if we choose a good step size, we can't differentiate the global from a local minima.

Neither of those problems can be solved completely. Nevertheless there are advanced optimization algorithms, which automatically adapt the step size and make the optimization sensitive to avoid saddle points. Moreover, experience has shown that the performance in local minima can be sufficient, making it unnecessary to securely reach the global minima.

2.3.3. Optimization Algorithms based on Gradient Descent

We now formally introduce some common algorithms used for Deep Learning models.

Definition 2.19 (Gradient Descent)**.** *The Gradient Descent update rule of parameters $\boldsymbol{\theta}$ with respect to a loss $\mathcal{L}$ and with gradient $\mathbf{g}_t = \nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}_t)$ at step t writes:*

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \mathbf{g}_t \tag{2.24}$$

*The factor η is named **learning rate**. As mentioned before the setting of this value is crucial for the convergence reachability and speed.*

Definition 2.20 (RMSProp[TH12])**.** *The Root Mean Square Propagation update rule of parameters $\boldsymbol{\theta}$ with respect to a loss $\mathcal{L}$ and with gradient $\mathbf{g}_t = \nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}_t)$ at step t writes:*

$$\begin{aligned} \mathbf{v}_{t+1} &= 0.9\mathbf{v}_t + 0.1\mathbf{g}_t^2 \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \frac{\eta}{\sqrt{\mathbf{v}_{t+1} + \epsilon}}\mathbf{g}_t \end{aligned} \tag{2.25}$$

The factor $\mathbf{v}_t$ is the running average over past squared gradients and therefore a local approximation for the mean squared gradient $\mathbb{E}[g^2]_t$. Its usage adapts the learning rate for each parameter individually, performing larger updates for infrequent and smaller updates for frequent parameters, which helps to avoid saddle points. Typical initialization sets $\mathbf{v}_0 = 0$ and $\eta = 0.001$.

Definition 2.21 (Adam[Die15])**.** *The Adaptive Moment Estimation update rule of parameters $\boldsymbol{\theta}$ with respect to a loss $\mathcal{L}$ and with gradient $\mathbf{g}_t = \nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}_t)$ at step t writes:*

$$\begin{aligned} \mathbf{m}_{t+1} &= \varphi_1\mathbf{m}_t + (1-\varphi_1)\mathbf{g}_t \\ \mathbf{v}_{t+1} &= \varphi_2\mathbf{v}_t + (1-\varphi_2)\mathbf{g}_t^2 \\ \hat{\mathbf{m}}_{t+1} &= \frac{\mathbf{m}_{t+1}}{1-\varphi_1^t} \\ \hat{\mathbf{v}}_{t+1} &= \frac{\mathbf{v}_{t+1}}{1-\varphi_2^t} \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \frac{\eta}{\sqrt{\mathbf{v}_{t+1}} + \epsilon}\mathbf{m}_{t+1} \end{aligned} \tag{2.26}$$

In addition to adapting the learning rate for each individual parameter, this method also replaces the gradient with a moving average. This tweak is called momentum, because it accelerates the

movement into an average direction and prevents unnecessary oscillation. Beyond that, the first and second moment estimates, approximated by the running averages, get bias corrected. Typical initialization sets $\mathbf{m}_0, \mathbf{v}_0 = \mathbf{0}$, $\eta = 0.001$,$\varphi_1 = 0.9$, $\varphi_2 = 0.999$ *and* $\epsilon = 10^{-8}$.

Note that all optimization algorithms require to set at least one further hyper-parameter (e.g. learning rate).

2.3.4. Stochastic Gradient Descent

While the optimization algorithms execute after the backward pass, altering the input samples x and y right at the beginning of the forward pass, also changes the path of our optimization.

Definition 2.22 (Batch Gradient Descent)**.** *Batch Gradient Descent updates the parameters* $\boldsymbol{\theta}$ *with respect to a loss* $\mathcal{L}$ *and with gradient* $\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}_t; X, Y)$ *using the complete input and output* X, Y *at step* t *at once:*

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \mathbf{g}_t \nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}_t; x, y) \tag{2.27}$$

Each samples contribution is summed up in the final loss, so all are treated equally. This approach shows low variance in the training curve.

Definition 2.23 (Stochastic Gradient Descent[TH12])**.** *Stochastic Gradient Descent updates the parameters* $\boldsymbol{\theta}$ *with respect to a loss* $\mathcal{L}$ *and with gradient* $\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}_t; x_i, y_i)$ *using the a single sample out of the input and output* X, Y*:*

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \mathbf{g}_t \nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}_t; x, y) \tag{2.28}$$

The gradient for each sample can be very different. This higher variance can help the algorithm to evade local minima. But the high variance leads to strong oscillations which impedes efficient convergence.

Definition 2.24 (Mini-batch Gradient Descent)**.** *Mini-batch Gradient Descent updates the parameters* $\boldsymbol{\theta}$ *with respect to a loss* $\mathcal{L}$ *and with gradient* $\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}_t; x_{i:i+\mu}, y_{i:i+\mu})$ *using a mini-batch of* μ *samples out of the input and output* X, Y*:*

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \mathbf{g}_t \nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}_t; x_{i:i+\mu}, y_{i:i+\mu}) \tag{2.29}$$

The mini-batch-size μ *determines the variance between batches and can heavily influence the convergence behavior. Batch and Stochastic Gradient Descent are thereby the extreme cases of setting* μ *to the total amount of samples* N *or* 1 *respectively.*

We conclude that the way we commit samples to a backpropagation step, changes the behavior of convergence and represents another hyper-parameter, namely the mini-batch-size.

2.4. Matrix Properties

We recap, that tasks with strong distortions in its inputs, require to scale up the number of hidden units m. The consequent growth of the m-by-m recurrent weight matrix can make an RNN intractable. The approach of this thesis suggests the restriction to use a recurrent weight band matrix. RNNs repeatedly multiply the recurrent weight matrix during forward as well as backward pass. Depending on its determinant, matrix multiplication can shrink or expand a volume, or in our case a signal. Longer sequences require more repetitions of multiplication. Without further input, the signal could therefore explode or vanish completely. Restrictions in the matrix properties alter this outcome. All in all, we have to explore, how different matrix properties will influence tractability and complexity of the model. We thereby have to include the gradient of the matrix multiplication into our analysis, since its tractability is equally important. In the following discussion we suppose that A is an m-by-m squared matrix and B and m-by-n matrix, corresponding to the recurrent weight matrix respectively the batched recurrent hidden states matrix of our RNN.

2.4.1. Unrestricted Matrix

Features

An unrestricted m-by-m matrix A can have any rank, determinant or eigenvalues. Used in classical RNNs they are initialized with identity or Gaussian random values. The former guaranteeing full rank together with determinant and hidden values of 1. Using the Gram-Schmidt Process the random matrix can be orthogonalized to the same properties. The expressiveness and the necessary storage increases linearly with the number of parameters and therefore squared in the matrix size $O(m^2)$. Multiplication with an m-by-n matrix B, results in an m-by-n matrix C. Having the definition of matrix multiplication 2.5 in mind, we conclude that we need m multiplications and m summations for $m \cdot n$ values. We conclude that multiplication requires $2nm^2$ operations, also experiencing squared growth.

Gradient of Multiplication

After the pre-work made in 2.22, we know, that for operations involving two variables, we have to find derivatives for $\frac{\partial \mathcal{L}(C)}{\partial C_{i,j}} \frac{\partial C_{i,j}}{\partial A_{a,b}}$ and $\frac{\partial \mathcal{L}(C)}{\partial C_{i,j}} \frac{\partial C_{i,j}}{\partial B_{a,b}}$.

Theorem 2.25 (Gradients of unrestricted Matrix Multiplication)**.** *Multiplication of unrestricted matrices $C = AB$ within the computational graph of the loss $\mathcal{L}(C)$ requires the computation of the following derivatives during Backpropagation:*

- $\frac{\partial \mathcal{L}(C)}{\partial A} = \frac{\partial \mathcal{L}(C)}{\partial C} B^T$
- $\frac{\partial \mathcal{L}(C)}{\partial B} = A^T \frac{\partial \mathcal{L}(C)}{\partial C}$

Proof. We insert the operation for matrix multiplication involving unrestricted matrices ($C_{i,j} = g(A,B)_{i,j} = \sum_{r=1}^{m} A_{i,k}B_{k,j} \overset{S.C.}{=} A_{i,k}B_{k,j}$) into the gradient equation:

$$\Big(\frac{\partial \mathcal{L}(C)}{\partial A}\Big)_{a,b} = \frac{\partial \mathcal{L}(C)}{\partial A_{a,b}} \overset{S.C.}{=} \frac{\partial \mathcal{L}(C)}{\partial C_{i,j}} \frac{\partial A_{i,k}B_{k,j}}{\partial A_{a,b}} = \frac{\partial \mathcal{L}(C)}{\partial C_{i,j}} \frac{\partial A_{i,k}}{\partial A_{a,b}} B_{k,j} = \frac{\partial \mathcal{L}(C)}{\partial C_{i,j}} \delta_{ia}\delta_{kb}B_{k,j} = \frac{\partial \mathcal{L}(C)}{\partial C_{a,j}} B_{b,j} = \frac{\partial \mathcal{L}(C)}{\partial C_{a,j}} B^T_{j,b} = \Big(\frac{\partial \mathcal{L}(C)}{\partial C} B^T\Big)_{a,b}$$

The matrix shapes $(m \times n) \cdot (n \times m)$ result in an m-by-m matrix matching the shape of A and its derivative. The derivation of the derivative of B is analogous:

$$\Big(\frac{\partial \mathcal{L}(C)}{\partial B}\Big)_{a,b} \overset{S.C.}{=} \frac{\partial \mathcal{L}(C)}{\partial C_{i,j}} A_{i,k} \frac{\partial B_{k,j}}{\partial B_{a,b}} = \frac{\partial \mathcal{L}(C)}{\partial C_{i,j}} A_{i,k}\delta_{ka}\delta_{jb} = \frac{\partial \mathcal{L}(C)}{\partial C_{i,b}} A_{i,a} = A^T_{a,i} \frac{\partial \mathcal{L}(C)}{\partial C_{i,b}} = \Big(A^T \frac{\partial \mathcal{L}(C)}{\partial C}\Big)_{a,b}$$

Now the matrix shapes are $(m \times m) \cdot (m \times n)$ result in an m-by-n matrix matching the shape of B and its derivative. □

Analyzing the necessary matrix products we find two standard matrix multiplications both growing squared $O(m^2)$.

2.4.2. Factorized Matrix

Definition 2.26 (Rank Factorization)**.** *Suppose A is an m-by-n matrix of rank r. A product $A = A^{(1)}A^{(2)}$ is called rank factorization or rank decomposition of A, where $A^{(1)}$ is an m-by-r matrix and $A^{(2)}$ is an r-by-n matrix.*

Features

The factoring dimension r is in upper bound for the rank of A. The multiplication of an m dimensional state with A will project it in an r dimensional subspace. Opposed to our goal, to increase the hidden state to bear more information, projecting it into a very small subspace would erase much of its information. [CJ17] showed that each hidden unit stores exactly one real value of the input history. Using a factorized matrix will limit the capacity to store only r real values. $A^{(1)}$ and $A^{(2)}$ have mr parameters each and necessary storage therefore increases linearly $O(m)$. The matrix multiplication AB is split in two separate multiplications $A^{(1)}A^{(2)}B$. First we carry out $A^{(2)}B$ which results in an r-by-n matrix and takes $2mrn$ operations. The result is then multiplied with $A^{(1)}$ requiring another $2rnm$ operations. We conclude that the number of operations grows linearly $O(m)$, but only if r is independent of the growth of m. The number of factoring dimensions r, becomes a switch between tractability and capacity. We expect this to become a strong limitation of its performance when a long input history is required.

Gradient of Multiplication

In the same manner as for unrestricted matrices 2.25, we derive the gradients for the factorized case.

Theorem 2.27 (Gradients of Factorized Matrix Multiplication)**.** *Multiplication of a factorized with an unrestricted matrix $C = A^{(1)}A^{(2)}B$ within the computational graph of the loss $\mathcal{L}(C)$ requires the computation of the following derivatives during Backpropagation:*

- $\frac{\partial \mathcal{L}(C)}{\partial A^{(1)}} = \frac{\partial \mathcal{L}(C)}{\partial C} B^T A^{(2)T}$
- $\frac{\partial \mathcal{L}(C)}{\partial A^{(2)}} = A^{(1)T} \frac{\partial \mathcal{L}(C)}{\partial C} B^T$
- $\frac{\partial \mathcal{L}(C)}{\partial B} = A^{(2)T} A^{(1)T} \frac{\partial \mathcal{L}(C)}{\partial C}$

Proof. We insert the operation for factorized matrix multiplication ($C_{i,j} = g(A^{(1)}, A^{(2)}, B)_{i,j} \overset{S.C.}{=} A^{(1)}_{i,k} A^{(2)}_{k,l} B_{l,j}$) into the gradient equation:

$$\left(\frac{\partial \mathcal{L}(C)}{\partial A^{(1)}}\right)_{a,b} = \frac{\partial \mathcal{L}(C)}{\partial A^{(1)}_{a,b}} \overset{S.C.}{=} \frac{\partial \mathcal{L}(C)}{\partial C_{i,j}} \frac{\partial A^{(1)}_{i,k} A^{(2)}_{k,l} B_{l,j}}{\partial A^{(1)}_{a,b}} = \frac{\partial \mathcal{L}(C)}{\partial C_{i,j}} \frac{\partial A^{(1)}_{i,k}}{\partial A^{(1)}_{a,b}} A^{(2)}_{k,l} B_{l,j} =$$
$$= \frac{\partial \mathcal{L}(C)}{\partial C_{i,j}} \delta_{ia}\delta_{kb} A^{(2)}_{k,l} B_{l,j} = \frac{\partial \mathcal{L}(C)}{\partial C_{a,j}} A^{(2)}_{b,l} B_{l,j} = \frac{\partial \mathcal{L}(C)}{\partial C_{a,j}} B^T_{j,l} A^{(2)T}_{l,b} = \left(\frac{\partial \mathcal{L}(C)}{\partial C} B^T A^{(2)T}\right)_{a,b}$$

The matrix shapes $(m \times n) \cdot (n \times m) \cdot (m \times r)$ result in an m-by-r matrix matching the shape of $A^{(1)}$ and its derivative. The derivation of the derivative of $A^{(2)}$ is analogous:

$$\left(\frac{\partial \mathcal{L}(C)}{\partial A^{(2)}}\right)_{a,b} = \frac{\partial \mathcal{L}(C)}{\partial C_{i,j}} A^{(1)}_{i,k} \frac{\partial A^{(2)}_{k,l}}{\partial A^{(2)}_{a,b}} B_{l,j} = \frac{\partial \mathcal{L}(C)}{\partial C_{i,j}} A^{(1)}_{i,k} \delta_{ka}\delta_{lb} B_{l,j} = \frac{\partial \mathcal{L}(C)}{\partial C_{i,j}} A^{(1)}_{i,a} B_{b,j} =$$
$$= A^{(1)T}_{a,i} \frac{\partial \mathcal{L}(C)}{\partial C_{i,j}} B^T_{j,b} = \left(A^{(1)T} \frac{\partial \mathcal{L}(C)}{\partial C} B^T\right)_{a,b}$$

Now the matrix shapes $(r \times m) \cdot (m \times n) \cdot (n \times m)$ result in an r-by-m matrix matching the shape of $A^{(2)}$ and its derivative. For the last derivative we can treat $A^{(1)}A^{(2)}$ as constant and therefore just insert them for A into the result of the unrestricted case: $A^{(2)T}A^{(1)T}\frac{\partial \mathcal{L}(C)}{\partial C}$. Checking the matrix shapes $(m \times r) \cdot (r \times m) \cdot (m \times n)$ we find the correct m-by-n of B and its derivative. □

Analyzing the necessary matrix products, we find three standard matrix multiplications, all growing linearly $O(m)$.

2.4.3. Orthogonal and Unitary Matrix

Orthogonal matrices were solely used for initialization of RNNs [HA16]. As a consequence, this property is lost after the first step of backpropagation. The RNN then

inherits the same complexity and tractability as in the unrestricted case. Unitary matrices on the other hand were used as fixed restriction of the recurrent weight matrix [AA16][WT16][VC17]. All different implementations thereby share, that they require a complex recurrent weight matrix, which changes complexity and tractability.

Definition 2.28 (Orthogonal Matrix)**.** *Suppose A is an m-by-m square matrix with body $\mathbb{R}$. Then A is orthogonal if the transpose A^T is also its inverse: $A^TA = AA^T = \mathbb{I}$.*

Definition 2.29 (Unitary Matrix)**.** *Suppose A is a n-by-n square matrix with body $\mathbb{C}$. Then A is unitary if the conjugate transpose A^* is also its inverse: $A^*A = AA^* = \mathbb{I}$.*

Theorem 2.30 (Orthogonal Matrices have Determinant of 1)**.** *Suppose A is an m-by-m orthogonal square matrix. Then the following statement always holds: $|A| = \pm 1$.*

Proof. Using the property 2.12 it is trivial to derive the determinant of the identity:

$$|\mathbb{1}| = \prod_{i=1}^{m} \lambda_i = \prod_{i=1}^{m} 1 = 1$$

Further using the property $|AB| = |A||B|$ and $|A^T| = |A|$ we find:

$$1 = |\mathbb{1}| = |A^TA| = |A^T||A| = |A||A| = |A|^2$$

□

Since this statement holds for any orthogonal matrix we further conclude that they have full rank and eigenvalues of ± 1.

Features

Multiplying a state with a unitary matrix acts as a rotation of the state. We will se in 3.2.2 that this property solves the vanishing and exploding gradient problem. Nevertheless it was already experimentally shown [VC17] that the restriction might be too strong. On its own, a unitary matrix is not capable to delete or even phase out information of a given state. To work computationally with complex numbers, the imaginary and real parts of the vector are concatenated and the vector doubles in size. The multiplication of an m-by-m complex unitary matrix with an m-by-n complex state matrix is then written as:

Definition 2.31 (Complex Matrix Multiplication)**.** *Suppose $A = Re(A) + i \cdot Im(A)$ is an m-by-r and $B = Re(B) + i \cdot Im(B)$ an r-by-n complex matrix. We define the complex matrix multiplication based on the real matrix multiplication as:*

$$\begin{pmatrix} Re(AB) \\ Im(AB) \end{pmatrix} = \begin{pmatrix} Re(A) & -Im(A) \\ Re(A) & Im(A) \end{pmatrix} \begin{pmatrix} Re(B) \\ Im(B) \end{pmatrix} \tag{2.30}$$

We fin that complex number multiplication is based on real number multiplication. The necessary storage doubles and the required number of operations quadruple. In the limit the computational costs will grow squared $O(m^2)$.

Gradient of Multiplication

Fully parameterized unitary matrix RNNs require a restriction of gradient descent to preserve its property [WT16]. Gradient steps have to move along the set of all possible unitary matrices, namely the Stiefel manifold [Tag11]. Without taking those additional computational costs into account, we find that backpropagation requires the gradient for an ordinary matrix multiplication with matrix sizes of $2m$-by-$2m$ respectively $2m$-by-n, showing squared growth $O(m^2)$.

2.4.4. Band Matrix

Definition 2.32 (Band Matrix)**.** *Suppose A is an m-by-m square matrix and φ a non-negative integer with $\varphi < m$, then A is a band matrix and φ its band-half-width if the following statement holds:*

$$A_{i,j} = 0 \quad \forall |i-j| > \varphi \tag{2.31}$$

Complexity

Opposed to orthogonal (always full rank) and factorized matrices (upper bound for rank is factoring dimension), band matrices can have full or lower then full rank. In case that all inside band values are non-zero, there exists a lower bound for the rank:

Theorem 2.33 (Lower bound for band matrix rank)**.** *Suppose A is an m-by-m band matrix with band-half-width φ and only non-zero values: $A_{i,j} \neq 0 \forall i, j \in 1, ..., m : |i-j| \leq \varphi$. Then:*

$$rank(A) \geq m - \varphi \tag{2.32}$$

Even though the complexity is obviously restricted with most of the band matrix values being zero, it stays close to full rank for comparable small band-half-widths. This comes in handy, since we want to avoid unnecessary limitation of the input history capacity. We will discuss further properties in 5.

Necessary Storage

An m-by-m band matrix still has m rows, but each row is only filled for a maximum of $2\varphi + 1$ columns. We therefore sum up the values to be $(2\varphi + 1)m - \varphi^2$, removing φ^2 indices laying outside the matrix. We see that the necessary storage grows linearly $O(m)$.

Matrix Multiplication Operations

For a clearer analysis we formally define the multiplication of a band-matrix with an unrestricted matrix.

Definition 2.34 (Band-Matrix Multiplication)**.** *Suppose A is an m-by-m band matrix with band-half-width φ and B is an m-by-n square matrix. Then the multiplication can be written as:*

$$(A \overset{b}{\circ} B)_{i,j} = \sum_{k=1}^{m} A_{i,k} B_{k,j} \delta_{|i-k| \leq \varphi} = \sum_{k=1}^{2\varphi+1} A_{i,k+i-\varphi} B_{k+i-\varphi,j} \tag{2.33}$$

Remark: For all non existing indices we input zero. We denote the band-matrix multiplication with the symbol $\overset{b}{\circ}$.

The above formula shows that we need $2\varphi + 1$ multiplications and summations for mn values, leading to $2(2\varphi + 1) \cdot mn$ operations. The number of operations for the multiplication therefore also grows linear $O(m)$.

Gradient of Multiplication

In the same manner as for unrestricted matrices 2.25, we derive the gradients for the band-matrix case.

Theorem 2.35 (Gradients of Band Matrix Multiplication)**.** *Multiplication of a band-matrix A with an unrestricted matrix B to form an unrestricted matrix $C = AB$ within the computational graph of the loss $\mathcal{L}(C)$ requires the computation of the following derivatives during Backpropagation:*

- $\frac{\partial \mathcal{L}(C)}{\partial A} = \frac{\partial \mathcal{L}(C)}{\partial C} \overset{2b}{\circ} B^T$
- $\frac{\partial \mathcal{L}(C)}{\partial B} = A^{\overset{b}{T}} \overset{b}{\circ} \frac{\partial \mathcal{L}(C)}{\partial C}$

Remark: We find that we need to implement two more band-matrix specific methods: band-matrix transpose $()^{\overset{b}{T}}$ and matrix multiplication to band-matrix $\overset{2b}{\circ}$.

Proof. We insert the operation for band matrix multiplication ($C_{i,j} = g(A,B)_{i,j} = \sum_{k=1}^{m} A_{i,k} B_{k,j} \delta_{|i-k| \leq \varphi} \overset{S.C.}{=} A_{i,k} B_{k,j} \delta_{|i-k| \leq \varphi}$) into the gradient equation:

$$\begin{aligned}\Big(\frac{\partial \mathcal{L}(C)}{\partial A}\Big)_{a,b} = \frac{\partial \mathcal{L}(C)}{\partial A_{a,b}} \overset{S.C.}{=} \frac{\partial \mathcal{L}(C)}{\partial C_{i,j}} \frac{\partial A_{i,k}}{\partial A_{a,b}} B_{k,j} \delta_{|i-k| \leq \varphi} = \frac{\partial \mathcal{L}(C)}{\partial C_{i,j}} \delta_{ia} \delta_{kb} \delta_{|i-k| \leq \varphi} B_{k,j} = \\ \frac{\partial \mathcal{L}(C)}{\partial C_{a,j}} B_{b,j} \delta_{|a-b| \leq \varphi} = \frac{\partial \mathcal{L}(C)}{\partial C_{a,j}} B^T_{j,b} \delta_{|a-b| \leq \varphi} = \Big(\frac{\partial \mathcal{L}(C)}{\partial C} \overset{2b}{\circ} B^T\Big)_{a,b}\end{aligned}$$

The matrix shapes $(m \times n) \cdot (n \times m)$ result in an m-by-m matrix restricted to zero values if $|a - b| > \varphi$. This means that we need an operation which calculates matrix multiplication for only those values which are necessary for a band. The derivation of the derivative of B starts analogous:

$$\begin{aligned}\Big(\frac{\partial \mathcal{L}(C)}{\partial B}\Big)_{a,b} \overset{S.C.}{=} \frac{\partial \mathcal{L}(C)}{\partial C_{i,j}} A_{i,k} \frac{\partial B_{k,j}}{\partial B_{a,b}} \delta_{|i-k| \leq \varphi} = \frac{\partial \mathcal{L}(C)}{\partial C_{i,j}} A_{i,k} \delta_{ka} \delta_{jb} \delta_{|i-k| \leq \varphi} = \\ \frac{\partial \mathcal{L}(C)}{\partial C_{i,b}} A_{i,a} \delta_{|i-a| \leq \varphi} = A^T_{a,i} \frac{\partial \mathcal{L}(C)}{\partial C_{i,b}} \delta_{|a-i| \leq \varphi} = \Big(A^T \overset{b}{\circ} \frac{\partial \mathcal{L}(C)}{\partial C}\Big)_{a,b}\end{aligned}$$

□

The derivative of B requires the multiplication of an m-by-m band-matrix with and m-by-n unrestricted matrix, which as derived earlier comes down to $2(2\varphi+1)\cdot mn$ operations therefore growing linear $O(m)$. To calculate the derivative of A we need a new operation:

Definition 2.36 (Matrix to Band-Matrix Multiplication)**.** *Suppose A and B are unrestricted m-by-n respectively n-by-m matrices. Their m-by-m matrix product is now restricted to be a band matrix with band-half-width φ:*

$$(A \overset{2b}{\circ} B)_{i,j} = \sum_{k=1}^{m} A_{i,k}B_{k,j}\delta_{|i-j|\leq\varphi} \overset{S.C.}{=} A_{i,k}B_{k,j}\delta_{|i-j|\leq\varphi} \tag{2.34}$$

If we would use standard matrix multiplication and subsequently set all out-of-band values to zero, we would need n summations and n multiplications for mm values resulting in $2n^2 \cdot m^2$ operations, leading to a squared growth of computational costs $O(m^2)$. Including this operation into our computational graph would be crucial, since it would be the bottleneck of our overall performance. As a consequence, writing an efficient implementation of the band-matrix matrix multiplication would be ineffective without an effective implementation of matrix to band multiplication. If we only calculate the in-band-values, we only need $(2\varphi+1)m$ values instead of m^2, and the overall algorithm will stay tractable, with computational costs growing only linearly $O(m)$.

3. Deep Learning Prerequisites

Deep learning is an area of machine learning, which offers a wide selection of successfully applied algorithms based on the same principles. The development of deep learning was heavily driven by building layered models, which propagate down the input information layer-per-layer until the loss is reached. Subsequently, all parameters are updated using backpropagation [RG86]. Making the models "deep", describes the approach of adding more and more layers. To keep the deeper models trainable, useful initializations [Hin06] and other tweaks were necessary [Sch15].

3.1. Neural Networks

3.1.1. Intuition

The term neural network stems from the essential use of linear combinations to propagate the information from one layer to the next. The information vector of a layer is called hidden state, and its dimensions are called hidden units. The matrix which connects two layers, adds dependencies between the different hidden units. To break the linearity of those linear transformation, a point-wise non-linear function is applied (typically a sigmoid $\sigma(x) = 1/(1 + e^{-x})$). This turns out to be crucial to entangle arbitrary spaces respectively to classify tangled inputs [Cyb89][HM89]. Moreover, this gives a first argument to apply more then one layer, since the capacity of each layer to entangle information in a space is limited. Using a finite number of units and layers, it can be proven that neural networks are universal approximators, or universal Turing machines, in case of RNNs. The original usage of sigmoid functions restricted the propagated values to lay between zero and one, giving rise to the comparison with the activation signal in the brain. Consequently, the dimensions of a hidden state could be interpreted as a neuron receiving the combined activation signal of the neurons of the previous layer. In this work we will focus on RNNs, which are especially designed to work on sequential data. We continue to introduce the feed forward neural network architecture, which build the basis for RNNs and other deep learning models.

3.1.2. Feed Forward Neural Networks

The basis of feed forward neural networks are densely connected layers.

Definition 3.1 (Densely Connected Layer)**.** *Suppose $h^{(l-1)} \in \mathbb{R}^n$ is the hidden state of layer $l-1$ respectively the model input. The subsequent layer is called densely connected, if its hidden*

state $h^{(l)} \in \mathbb{R}^m$ depends on all values $h^{(l-1)}$, fulfilling the following equation:

$$h^{(l)} = a(Wh^{(l-1)} + b) \tag{3.1}$$

Thereby:

- *$a(\cdot)$ is a point-wise applied function, usually sigmoid $\sigma(\cdot)$ or $tanh(\cdot)$*
- *W is an m-by-n matrix applying a linear map $\mathbb{R}^n \to \mathbb{R}^m$*
- *$b \in \mathbb{R}^m$ is a bias vector, necessary add translations of the origin to the operation*

Storage and operations required to calculate densely connected layers grow linearly in the number of hidden units $O(m)$.

3.2. Recurrent Neural Networks

RNNs are an extension of feed forward models to treat sequential data. They add the dependency between hidden units of different time steps. Hidden states of subsequent time steps, depend on previous time steps. Propagating the hidden states through time, the model achieves the capacity to store input history. Due to the recurrence relation it is common to speak of **cells** instead of layers. Note that depending on the context, RNN refers to cell as well as to the whole model.

3.2.1. Original Architecture

Definition 3.2 (Recurrent Neural Network Cell)**.** *Suppose $h_t^{(l-1)} \in \mathbb{R}^n$ is the hidden state of layer $l-1$ respectively the model input at time step t. An RNN cell receiving $h_t^{(l-1)}$ as input and giving $h_t^{(l)} \in \mathbb{R}^m$ as an output depends on the past hidden state of the same layer $h_{t-1}^{(l)}$ using the following equation:*

$$h_t^{(l)} = a(Wh_t^{(l-1)} + Ah_{t-1}^{(l)} + b) \tag{3.2}$$

Remark: Except the recurrent m-by-m weight matrix A, all components are identical to the feed forward case. The first time step $t = 1$ requires hidden state $h_0^{(l)}$ as input. $h_0^{(l)}$ is usually initialized to be zero and treated as a trainable parameter.

Having the matrix properties in mind, we find that the necessary storage and operations grow squared in the number of hidden units $O(m^2)$. The recurrent weight matrix turns out to be the bottleneck in the algorithms' performance.

3.2.2. Vanishing and Exploding Gradient

Repeated multiplication of matrices can lead to vanishing or exploding signals, as already discussed in the mathematical prerequisites. This turns out to be a characteristic

problem for RNNs. If we input the recurrence relation of past hidden states into the RNN cell equations, we can see how this problem affects the forward pass:

$$h_t^{(l)} = a(Ah_{t-1}^{(l)} + ...) = a(Aa(A...a(Ah_0^{(l)} + ...)...))$$

Depending on the properties of A, the values of older hidden states are shrunk or expanded, determining their influence on later time steps. To find the right balance, A requires the updates through backpropagation. To analyse the backward pass we simplify our model: the loss depends only on the output of the last time step $\mathcal{L}(h_T^{(l)})$. To update the parameters of A we need its gradient:

$$\frac{\partial \mathcal{L}(h_T^{(l)})}{\partial A_{a,b}} = \frac{\partial \mathcal{L}(h_T^{(l)})}{\partial h_T^{(l)}} \frac{\partial h_T^{(l)}}{\partial A_{a,b}}$$

To emphasize the role of A we ignore terms independent of it. Further focusing on the second derivative $\frac{\partial h_T^{(l)}}{\partial A_{a,b}}$ we find a recurrence relation for the gradient:

$$\frac{\partial h_T^{(l)}}{\partial A_{a,b}} = \frac{\partial a(Ah_{T-1}^{(l)} + ...)}{\partial A_{a,b}} = \frac{\partial a(...)}{\partial ...}(\frac{\partial A}{\partial A_{a,b}} h_{T-1}^{(l)} + A\frac{\partial h_{T-1}^{(l)}}{\partial A_{a,b}}) = ...(... + A\frac{\partial h_{T-1}^{(l)}}{\partial A_{a,b}})$$

We continue to input the recurrence relation of past hidden states as done for the Forward Pass:

$$\frac{\partial h_T^{(l)}}{\partial A_{a,b}} = ...(... + A(...(... + A\frac{\partial h_{T-2}^{(l)}}{\partial A_{a,b}}))) = ...(... + A(...(... + A(...(... + A\frac{\partial h_{T-3}^{(l)}}{\partial A_{a,b}})))))$$

We find the following relationship between the derivatives of time step T and earlier time steps $t < T$:

$$\frac{\partial h_T^{(l)}}{\partial A_{a,b}} = ... + ...A^t \frac{\partial h_{T-t}^{(l)}}{\partial A_{a,b}}$$

We conclude that, the further a hidden state lies in the past, the stronger its derivative is influenced by the properties of A. For $|A| > 1$ derivatives of past hidden states gain an exploding signal, while for $|A| < 1$ they eventually vanish completely. As a consequence, the parameter updates themselves end up to be strongly influenced by the properties of A, either biased towards ignoring the past respectively the present information. To overcome this effect was a major drive for the development of new models. In this approaches we find two major categories:

- **restricted models** which initialize and/or restrict the recurrent weight matrix to have suitable properties.
- **gated models** which change the mathematical architecture of the RNN to alter the information flow during the backward pass.

3.3. Restricted Recurrent Neural Networks

Restriction of the recurrent weight matrix to specific properties can solve the vanishing or exploding gradient problem. With the limitation to a only a subspace of matrices, we also limit the solution space for our problem. It could therefore be of advantage to loosen the restriction at a later and more stable stage of training. Initialization is an extreme case of this approach, loosening the restriction after the first backpropagation step.

3.3.1. Identity

[LN15] proposed the use of classical RNN methods with identity initialization. The identity is obviously orthogonal and solves the exploding and vanishing gradient problem, as long as it is kept constant. To avoid that the non-linearity repeatedly decreases the signal, they suggest the use of rectifier linear units ReLU ($f_{relu}(x) = max(0, x)$).

Definition 3.3 (IRNN)**.** *An IRNN is in its architecture identical to an ordinary RNN, with the condition that the recurrent weight matrix has to be initialized with identity and the recurrent transfer function has to be a ReLU.*

They could show that trainability compared to RNNs increases. Holding the recurrent weight matrix fixed for more then the first iteration could further increase the performance. As soon as the matrix is trainable we will face computational costs growing squared in the number of hidden units $O(m^2)$.

3.3.2. Factorized

[LV16] and [KG17] propose the use of a low rank factorized recurrent weight matrix for RNNs respectively LSTMs.

Definition 3.4 (Factorized RNN)**.** *A factorized RNN is in its architecture identical to an ordinary RNN, with the condition that the recurrent weight matrix is low rank factorized into two matrices.*

The factoring dimension r will be the upper bound of the recurrent weight matrix rank. Old hidden states arriving with dimensionality m will get mapped into a lower dimensional subspace with dimensionality r. As mentioned before, this will limit our models capacity to store input history to r real values [CJ17]. If the task requires high input history capacity, we expect factorized models to fail. The current input on the other hand, which is multiplied with the input weight matrix, gets projected in an m dimensional space, which could be of advantage if the incoming input is very high dimensional. As derived in 2.4.2, the necessary storage and the number of operations grow linearly in the number of hidden units $O(m)$.

3.3.3. Orthogonal

The work of [HA16] generalizes the identity initialization of the IRNN to all orthogonal matrices. Moreover, they drop the ReLu transfer function, but instead exclude the recurrent weight matrix operation from the sigmoidal non-linearity:

Definition 3.5 (ORNN)**.** *An ORNN is in its architecture identical to an ordinary RNN, with the condition that the recurrent weight matrix has to be initialized orthogonally and the transfer function has to exclude the recurrent weight matrix operation:*

$$\sigma(Wh_t^{(l-1)} + b) + Ah_{t-1}^{l} \tag{3.3}$$

The initial conditions of the model therefore solve the vanishing and exploding gradient problem. Performance was reported for smaller artificial problems, with no need to use a high number of hidden units. Computational costs grow squared in the number of hidden units $O(m^2)$.

3.3.4. Unitary

Three different implementations for RNNs with the restriction of a unitary recurrent weight matrices are available, increasing in generality. The most restricted case works on a subspace of all possible unitary matrices, leading to lower complexity but an increased computational performance[AA16]. They report storage growth of $O(m)$ and operational growth of $O(m \log(m))$, but don't offer experiments requiring a high number of hidden units. [WT16] generalized the model to use a full parametrization of unitary matrices. While training they keep the matrix inside the set of all possible unitary matrices, namely the Stiefel manifold. Using insights from [Nis05] and [Tag11] they propose geodesic gradient descent, a restriction that gradient descent steps have to move along the Stiefel manifold. This full capacity unitary RNN increased in performance on standard tasks and the computational costs for storage and operations grow squared in the number of hidden units $O(m^2)$ (also at least four times as much as for standard RNNs). Finally [VC17] generalized full capacity unitary RNNs even further and allowed deviations from the unitary matrix restriction. The strength of deviations was controlled by an extra hyper-parameter named margin. Again an improvement in performance in standard tasks was reported. The slightly higher computational costs still grow squared $O(m^2)$. All models were tested on the Copying and Addition task [HS97] as well as on sequential MNIST. Although being unaligned tasks, the type of memory they need is very specific and in fact does not require high capacity to store input history. A comparison with our model is at this point still outstanding.

Definition 3.6 (URNN)**.** *A URNN is in its architecture identical to an ordinary RNN, with the condition that the recurrent weight matrix is restricted to be a unitary matrix. This can be achieved by design (restricted URNN) or geodesic gradient descent (full URNN). Also loosening up the restriction to only a degree of unitarity is possible (margin URNN).*

3.4. Gated Recurrent Neural Networks

The already 1997 proposed long-short term memory network (LSTM) [HS97] gave rise to an independent development of gated recurrent neural networks. Besides LSTMs we will discuss gated recurrent units (GRU) [CB14]. Crucial in both models is the fact that at least one recurrent hidden state is propagated through time without facing matrix multiplication or non-linearities. Only basic operations like element-wise addition and multiplication of so called gates will change that hidden state. This trick breaks up the repeated application of the recurrent weight matrix, leaving a non exploding or vanishing loophole for the forward and backward pass. [CJ17] argues that this strongly increases the trainability, but inherits the same task and input-history storage capacity per parameter respectively per hidden unit as traditional RNNs. Since gated RNNs are not the focus of this thesis, we recommend to read the original publications to gain further understanding of how they solve the vanishing and exploding gradient problem.

3.4.1. LSTM

The architecture of a typical LSTM cell can be seen in 3.1. Besides a gated structure, LSTM cells make use of a newly introduced recurrent state, called cell state [HS97]. This cell state fulfils the necessary condition of gated models: its information is only changed by element-wise addition and multiplication, excluding matrix multiplication and non-linear functions. The cell state enables a way during backpropagation to respect states laying longer in the past, leading to increased trainability from the start. The gates then organize to update and read out the cell state in a proper way to propagate it down to the next layer. The values of the sigmoidal gates ($\forall x : \sigma(x) \in]0, ..., 1[$) can be interpreted as filters, the values of the *tanh* candidate gate ($\forall x : \tanh(x) \in]-1, ..., 1[$) as update, being able to delete and write:

- The **forget gate** filters out information of the past cell state.
- The **input gate** balances the new information of the **candidate gate**, which will be written onto the cell state.
- The **output gate** reads out the information of the cell state.

With this interpretation and the visualization in 3.1 we now add the underlying model formulas:

Definition 3.7 (Long Short Term Memory Cell)**.** *Suppose $h_t^{(l-1)} \in \mathbb{R}^n$ is a hidden state or the model input at time step t. An LSTM Cell receiving $h_t^{(l-1)}$ as input and giving $h_t^{(l)} \in \mathbb{R}^m$ as an output depends on the past hidden state of the same layer $h_{t-1}^{(l)}$ as well as on the past cell state*

$c_{t-1}^{(l)}$ *using the following equations[HS97]:*

$$
\begin{aligned}
f_t &= \sigma(W_f^{rec} h_{t-1}^{(l)} + W_f^{inp} h_t^{(l-1)} + b_f) \\
i_t &= \sigma(W_i^{rec} h_{t-1}^{(l)} + W_i^{inp} h_t^{(l-1)} + b_i) \\
o_t &= \sigma(W_o^{rec} h_{t-1}^{(l)} + W_o^{inp} h_t^{(l-1)} + b_o) \\
\tilde{c}_t &= \tanh\left(W_{\tilde{c}}^{rec} h_{t-1}^{(l)} + W_{\tilde{c}}^{inp} h_t^{(l-1)} + b_{\tilde{c}}\right) \\
c_t^{(l)} &= f_t c_{t-1}^{(l)} + i_t \tilde{c}_t \\
h_t^{(l)} &= o_t \tanh c_t^{(l)}
\end{aligned}
$$

LSTMs afford the computation of four RNN Cells. Each recurrent matrix is m-by-m and turns out to be the storage and computational bottleneck. Hence computational costs grow squared in the number of hidden units $O(m^2)$ and are around 4 times higher then standards RNNs.

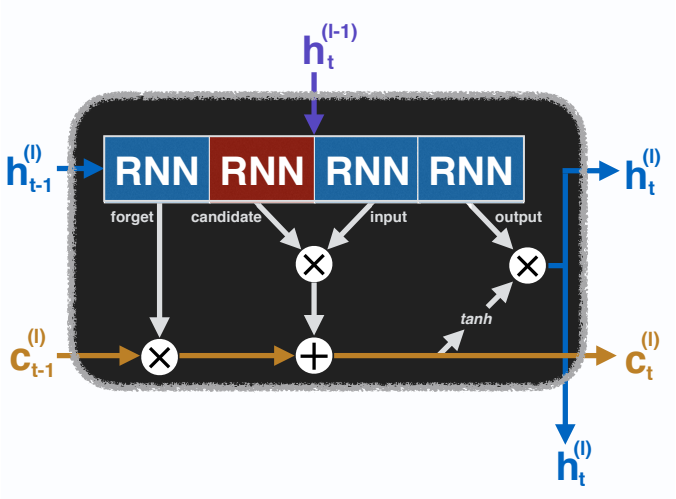

Figure 3.1.: Architecture of an LSTM Cell. The red RNN cell has a non-standard *tanh* non-linearity.

3.4.2. GRU

The gated recurrent unit architecture (developed 17 years after the LSTM) drops the extra cell state of the LSTM. Now the recurrent hidden state itself gets propagated through time, without facing matrix multiplication or non-linear functions. Compared to the LSTM gates, the functionality of the GRU gates can be explained the following way:

- The **reset gate** r_t filters out information of the past hidden state, preparing it for the candidate gate.

- The **candidate gate** $\tilde{c}_t$ takes the reset past hidden state as an input.
- The **update gate** u_t combines LSTMs forget and input gate. Though $(1 - u_t)$ is used to filter out information of the past hidden state and u_t to balance the information of the new candidate gate.

LSTM's output gate is dropped completely, due to the missing cell state. Note that compared to LSTMs, GRUs use a stacked application of RNNs to calculate the candidate.

Definition 3.8 (Gated Recurrent Unit Cell)**.** *Suppose $h_t^{(l-1)} \in \mathbb{R}^n$ is a hidden state or the model input at time step t. A GRU Cell receiving $h_t^{(l-1)}$ as input and giving $h_t^{(l)} \in \mathbb{R}^m$ as an output also depends on the past hidden state of the same layer $h_{t-1}^{(l)}$:*

$$
\begin{aligned}
r_t &= \sigma(W_r^{rec} h_{t-1}^{(l)} + W_r^{inp} h_t^{(l-1)} + b_r) \\
u_t &= \sigma(W_u^{rec} h_{t-1}^{(l)} + W_u^{inp} h_t^{(l-1)} + b_u) \\
\tilde{c}_t &= \tanh\Big(W_{\tilde{c}}^{rec}(r_t h_{t-1}^{(l)}) + W_{\tilde{c}}^{inp} h_t^{(l-1)} + b_{\tilde{c}}\Big) \\
h_t^{(l)} &= (1 - u_t) h_{t-1}^{(l)} + u_t \tilde{c}_t
\end{aligned}
$$

GRUs afford the computation of three RNN Cells. Each recurrent matrix is m-by-m and turns out to be the storage and computational bottleneck. Hence computational costs grow squared in the number of hidden units $O(m^2)$ and are around 3 times higher then standards RNNs.

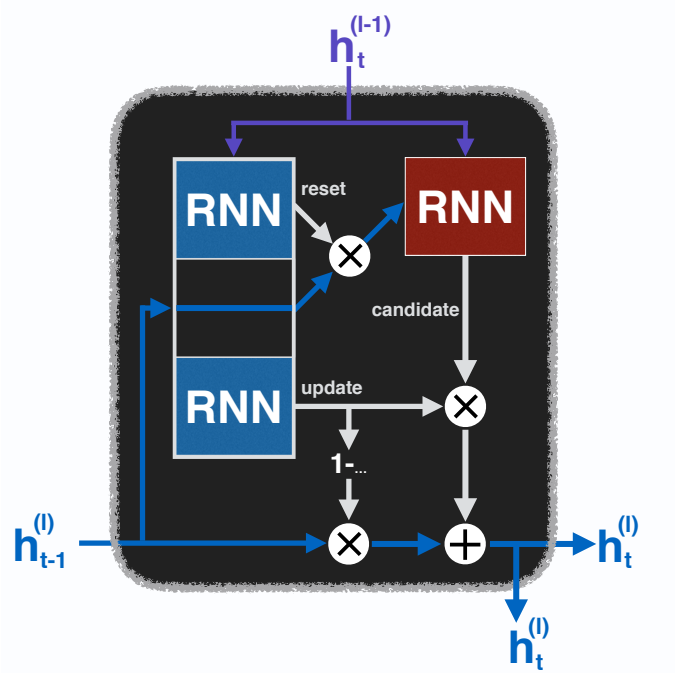

Figure 3.2.: Architecture of a GRU Cell. The red RNN cell has a non-standard *tanh* on-linearity.

4. Technical Prerequisites

In the previous chapters we identified matrix multiplication as computational bottleneck growing squared for RNNs in the number of hidden units $O(m^2)$. Linear algebra algorithms in general turn out to be very costly and require efficient implementation and special hardware to deal with. Lately especially high-definition computer graphics lead to big improvements in graphical processing units (GPUs), which are now the standard components to deal with parallelizable problems, including most linear algebra algorithms. Widespread software implementations like the basic linear algebra subprograms (BLAS) offer very efficient algorithms, also customized for different GPU architectures, e.g. the closed source CUBLAS library for nVidia GPUs. Unfortunately the BLAS/CUBLAS libraries do not contain the implementation of a band-matrix matrix multiplication kernel, which would have to take into account that most band-matrix values are zero. This required us to write our own implementation, based on the available open source project MAGMA [NS10]. We now give the technical background to understand our implementation in chapter 6. All details about GPUs may only apply to nVidia GPUs. This section is strongly based on nVidias *Programming Guide*[nVi17b] and *Best Practices Guide* [nVi17b] in the CUDA Toolkit documentation [nVi17c].

4.1. GPU

The development of GPUs was strongly driven to meet the demands of high-definition real-time 3D graphics, which in fact need efficient linear algebra algorithms to process pixels [nVi17b](1). CPU and GPU pursue different strategies to minimize latency (visualized in Figure 4.1): CPU architecture attempts to **minimize** latency **within** a thread, while GPU architecture **hides** latency **across** across threads, more precisely across groups of threads named warps [Bal11]. CPUs use branch prediction or speculative execution to cut waiting times within in a thread [nVi17b](2.1). GPUs use high memory bandwidth and multiple cores for multithreaded data-parallel processes. While modern hyper-threading CPUs can run up to 48 threads concurrently, modern GPU streaming multiprocessors can support up to 2048 threads concurrently, leading to more then 10240 active threads for GPUs with 5 streaming multiprocessors.

4.1.1. Microarchitecture

The hardware components host and device contain the CPU respectively GPU chip [nVi17b](4). The device is connected to the host with a fast PCI express (PCIe) port, over which data is exchanged between host and device memory. The GPU chip itself contains

multiple streaming multiprocessors (SMs), each containing many CUDA cores, shared memory and warp schedulers. As an example Figure 4.4 shows an SM with fermi architecture. To manage a large amount of threads at the same time, the GPU employs a unique architecture called Single-Instruction, Multiple-Thread (SIMT) [nVi17b](4.1). It therefore groups 32 threads together to form a warp and execute one common instruction at a time. Threads forming a warp should agree on their execution path. If they diverge though data-dependent conditional branches, exemplified in Figure 4.2, the performance of a warp decreases drastically. Moreover, the warp schedulers constantly switch between the warps, to keep the cores running. They are said to hide the latency of memory accesses and arithmetic pipelines behind each other, visualized in Figure 4.1. Note that microarchitectures are under permanent development. Hence, nVidia differentiates between so called compute capabilities [nVi17b](2.5).

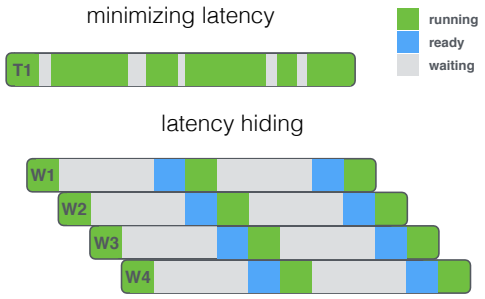

Figure 4.1.: Difference between latency minimizing and hiding.

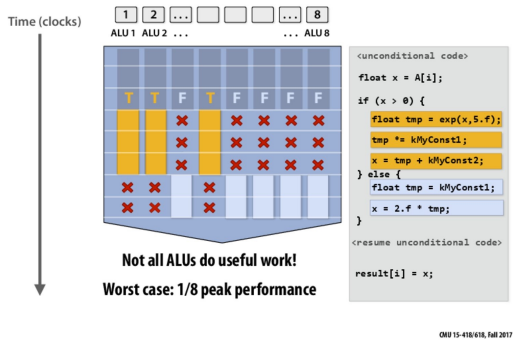

Figure 4.2.: Thread respectively warp divergence [MR15].

4.1.2. CUDA

CUDA is a C programming language extension to specify how a program should be executed on the GPU. Therefore a so called kernel is written, which defines the work each thread of the program has to accomplish. To make efficient use of shared memory, threads are grouped together to blocks [nVi17b](2.2). Thereby we should keep in mind that the GPU itself schedules warps, 32 threads at a time. Every thread has an ID *tid* on the basis of which they are grouped together into warps. Blocks on the other hand can be organized in a grid of blocks, which then should be evenly submitted to all available SMs. Figure 4.3 shows the hierarchy of grids, blocks and threads, together with the memory layer over which they communicate. We will learn in the chapter about performance how we should set the block size and grid dimensions, depending on our program and the GPU microarchitecture. Note that the latter dependency makes it necessary to customize kernels for the same program to different microarchitectures.

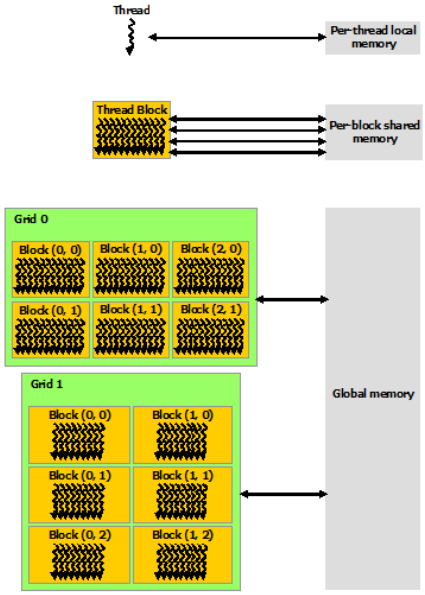

Figure 4.3.: Programming Model and Memory hierarchy [nVi17b].

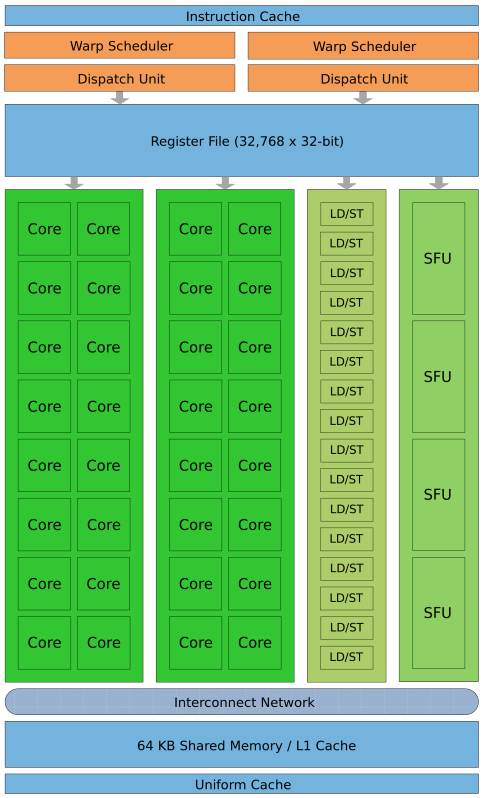

Figure 4.4.: Streaming multiprocessor fermi architecture [NVI09].

4.1.3. Performance Guidelines

To maximize parallel execution and hide as much latency as possible we have to respect utilization and memory throughput [nVi17b](5). In analogy, to increase the throughput of a water-tap we can turn it clockwise or counterclockwise. Now imagine ten taps, some of them depending on each other. We have to minimize all potential bottlenecks so that the water can flow. In the same way we have to think about our data, in our case the matrix parameters. How can they flow through our application, without getting stuck to much. For further details about instruction throughput we recommend [nVi17a](11). We confined ourselves to use single precision and didn't apply further optimizations.

Maximum Utilization

All components able to run concurrently should be utilized.
On device level, our blocks should spread evenly across all available SMs and keep them busy [**bibid**](5.2.2).
Modern SMs are capable to organize 2048 threads respectively 64 warps concurrently. On SM level it is therefore preferable to spread the work across all concurrently runnable

warps. If one warp has to wait for data or at a synchronization barrier, another ready-to-work warp will be scheduled. Only if enough warps are resident, latency hiding is happening. The number of active warps divided by all concurrently runnable warps is called occupancy [nVi17a], determining how effectively the hardware is kept busy. There are different factors which can limit occupancy. If the dimensions of the scheduled blocks are not multiples of the warp size, some warps are wasted with only a fraction of the 32 threads actually working. Moreover warps can block to much of the shared memory, leaving other warps waiting. Threads may exceed their maximum number of registers, requiring additional data transfers to local memory. The CUDA function *cudaOccupancyMaxPotentialBlockSize* checks the kernel with respect to all those conditions and calculates the block size for which the occupancy achieves its maximum. Nevertheless there are cases in which it is favorable to lower occupancy (running less warps/threads) to increase the amount of registers per thread [NS10].

Maximum Memory Throughput

Figure 4.3 shows the for us important memory layers of a GPU. In addition to the illustrated global, local and shared memory each thread has registers of a CUDA core available.
On the lowest level, data transfers between host and device should be limited [nVi17a](9.1), which is mostly a concern for the used deep learning framework itself. It should try to keep all repeatedly used parameters and data on the device while the algorithm is running.
On the next level, data transfers from global memory increase dramatically in speed if the data is coalesced across the warp [nVi17a](9.2.1.). E.g. let all threads in the warp carry out a data transfer from global into shared memory. If the memory access is coalesced then the values in global memory, corresponding to each thread, are sequential and aligned with the cache lines. Sequential means that the value of thread 0 lies next to the value of thread 1, and so on. Cache lines are a more substantial structure from which 128 bytes transfers with a single instruction are possible. Violation of coalesced memory leads in the worst case to 32 times more instructions.
Given that we use our matrix parameters multiple times during multiplication, it would be of big advantage to reuse them across threads. Shared memory is shared by all threads working in the same block and has a much higher bandwidth then global memory [nVi17a](9.2.2.). Further it is divided into equally sized memory banks that work simultaneously. But in case that two threads request the same bank, serialized loads are necessary and the data transfers slow down, which is called bank conflict.
If the compiler detects that a thread is exceeding the available registers, it shifts data to local memory. Even though local memory is dedicated to a single thread, it requires much longer transfer times (comparable to global). Its use should therefore be limited [nVi17a](9.2.3).

4.2. Matrix Multiplication Kernels

Matrix multiplication is a well known parallelization problem. Many tutorials explaining the GPU architecture use it as an instructive example [nVi17b](3.2.3). As already mentioned, the standard linear algebra libraries (BLAS/CUBLAS) unfortunately do not include the case of band-matrix matrix multiplication. Hence we tried to base our own implementation on available open source code. The state-of-the-art kernels of the CUBLAS library are closed source, therefore inaccessible. The MaxAs [Gra16] project analyzed the CUDA binaries and built an assembler, offering a new way to write highly efficient code in register level instructions. Based on their offered SGEMM kernel, our implementation achieved very high performance, but lead to problems on different microarchitectures, which were hard to maintain. The open source project MAGMA [NS10] offers a BLAS implementation in ordinary CUDA code. Since their optimizations were included into the CUBLAS kernels, it was not further developed. Based on the MAGMA single precision matrix-matrix multiplication kernel SGEMM, we were able to implement stable kernels for band-matrix matrix multiplications. We further explain MAGMA SGEMM together with our kernels in 6.

5. Band Recurrent Neural Networks

This thesis proposes a new type of RNN for which the recurrent weight matrix is restricted to be a band matrix.

Definition 5.1 (Band RNN)**.** *A Band RNN is in its architecture identical to an ordinary RNN, with the condition that the recurrent weight matrix is restricted to be a band matrix.*

The properties of band matrices were already discussed in 2.4.4, the architecture of restricted neural networks already in 3.3. In this chapter we want to take a closer look into the features our model inherits and what possible extensions it offers.

5.1. Tractability

The computational properties were already thoroughly discussed in 2.4.4. As we increase the number hidden units m, classical methods face squared growth in necessary matrix storage and multiplication operations $O(m^2)$. Band models on the other hand grow only linearly $O(m)$, making them much more tractable. The dynamic of increase in parameters compared to other models is visualized in 7.1. The effective speed of models was tested in a separate experiment shown in section 7.4. As a consequence of this increased tractability, we can increase the hidden units to very large sizes (in our experiments up to 5888).

5.2. Memorizing and forgetting

Compared to factorized or unitary models, the band model offers some interesting features. Factorized models fail to effectively use the high dimensional states. They are by definition rank deficient with the factoring dimension r being an upper bound. Band models will in most cases (for no inside band zeros) face the complete opposite, having a lower bound for their rank at $m - \varphi$. Since φ is usually small compared to m to keep the model tractable and we only lose φ dimensions at maximum, our models will have close to full rank. Unitary models on the other are restricted to have full rank, preserving the state dimensionality and blocking vanishing forward passes through eigenvalues of ± 1. But given that properties they have no way to erase or at least phase memory out of the hidden state themselves. They require the help of new inputs to delete old memory. Although band matrices will very likely have close to full rank, their eigenvalues are not restricted. That way the matrix itself is able to remove information from the hidden state. On the other hand, band matrix models will suffer the vanishing

gradient problem explained in section 3.2.2. As in other approaches [LN15][HA16] we tried to tackle this problem with clever initialization, which we will discuss in 5.7. Besides that, we argue that the behavior after a parameter update is less chaotic then in the unrestricted case, due to much less dependencies and complexity. The manifold of all possible band matrices is much smaller then the manifold of all possible squared matrices. With our experiments we try to give first evidence that band models suffer less of the vanishing gradient problem then other RNNs and are very robustly trainable.

5.3. Storage capacity shift

The work of [CJ17] proposed that model performance is driven by its capacity to store valuable task information into its parameters. Moreover they found all RNN models (also gated) to inherit exactly the same per parameter capacity and concluded that differences between the models are caused by trainability. Respective to the capacity to store input history they also found the same capacity over all models (1 real value per hidden unit). Finally they suggest that contrary to common belief, the capacity to store more input history is not a limiting factor of the model performance. Band matrices are designed completely opposed to this suggestion. They are able to increase the number of hidden units to large level, while possessing comparably less parameters. E.g. an LSTM with 1024 hidden units inherits the same amount of parameters then a $\varphi = 32$ band model with 55000 hidden units. We see that we can dramatically increase the capacity to store input history, while keeping the task storage capacity low. If the suggestions of [CJ17] hold this does not influence the overall model performance. We challenge that this result is in fact depending on the task itself. Some tasks might require low capacity to store input history, others might require a higher capacity. Our experiments with distorted inputs in long sequences strengthen this argument 7.4.

5.4. Comparison to a dynamical system

The recent work of [AK16] showed that to successfully simulate complex dynamical systems, short and long term interactions are necessary. Suppose our recurrent hidden vector is the state of a complex system. The multiplication of this state with a band matrix acts as local interaction in this state. Disregarding units in the state's head or tail, a single unit is only affected by 2φ units lying in its neighborhood, and independent of others. After a second application of this matrix the neighborhood already increased to 2φ, hence after an n-th to $n \cdot 2\varphi$. E.g. for $m = 4096$ and $\varphi = 32$, the model needs 64 steps to pass messages from one end to the other.

5.5. Combination with a grid

Inspired by the lack of long term interactions, we propose the combination of a band matrix with a sparse grid matrix. All dimensions which are part of the grid are then connected with each other, no matter how distant they lay in the hidden state vector. The multiplication with band-grid combination matrix will now also pass information via the grid dimensions.

Definition 5.2 (Grid Matrix)**.** *Suppose G is an m-by-m matrix, $\lambda \in 1, ..., m$ and $\mathcal{I} \subset 1, ..., m$ with $|\mathcal{I}| = \lambda$. We call G a grid matrix if the following statement holds:*

$$G_{i,j} = 0 \forall i, j \notin \mathcal{I} \tag{5.1}$$

Definition 5.3 (Band Grid Combination RNN)**.** *A Band Grid Combination RNN is in its architecture identical to an ordinary RNN, with the condition that the recurrent weight matrix is restricted to be of type $W_{rec} = A + G$ with A being a band and G being a grid matrix of the same size.*

λ arises as a new hyper-parameter and relieves φ of being the message passing limiting factor. E.g. for $m = 4096$ and $\lambda = \varphi = 32$, non-grid units need 3 steps to pass messages through the whole state. The first to interact with a neighboring grid unit, the second for an interaction between the grid units and a third step from the grid units to its local neighborhood. Computationally the format of G used in the grid matrix definition is not very effective. Due to associativity of matrix addition and multiplication we can first carry out the multiplications and add the results afterwards: $(A + G)B = AB + BG$. Using that most values of G are zero we store only the λ^2 non-zero values of G in a separate matrix $\hat{G}$. We then select the corresponding rows out of matrix B, which we call $\hat{B}$. We then carry out ordinary matrix multiplication of an λ-by-λ matrix and an λ-by-n matrix. The results we then add to the result of AB. That way the storage as well as the operations are completely independent of m and grow squared for the grid size $O(\lambda^2)$.

5.6. Closed Band RNN

Another possible extension to the band RNN is to fill in some values in the upper right and the lower left corner. Figure 5.1 visualizes that we can think of those parameters, as being the ones cut off by the matrix limits. We then add them back into the corners of the matrix. This way we connect hidden units lying in the head of the hidden state with units in its tail, closing a circular dependency. All hidden units now face the same amount of neighbors when multiplied with a matrix of this type. Moreover, the minimum number of applications to pass a message between all hidden units is cut in half, since the information jumps over the limits of the state from the bottom to the top and vice versa. Another interesting feature is the possibility to initialize this matrix with a shift matrix, which will be further discussed in section 5.7. We continue to give a formal definition of a close band matrix and a closed band model.

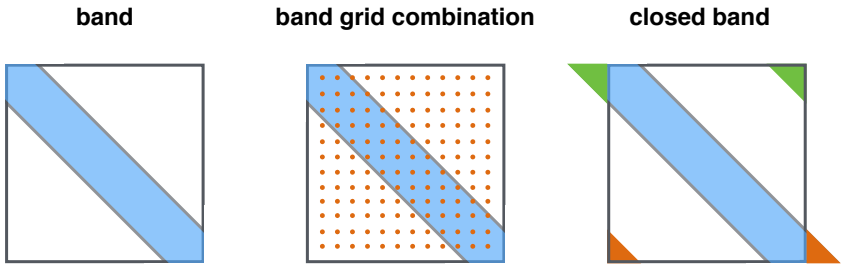

Figure 5.1.: Band matrix together with the grid and closed extension.

Closed Band Matrix Suppose A, L and U are m-by-m matrices and φ an integer with $0 \leq \varphi < m$. Then the sum $C = A + L + B$ is called Closed Band matrix if the following three properties hold:

$$\begin{aligned} A_{i,j} &= 0 \forall |i-j| > \varphi \\ L_{i,j} &= 0 \forall m - i + j > \varphi \\ U_{i,j} &= 0 \forall m - j + i > \varphi \end{aligned} \tag{5.2}$$

Definition 5.4 (Closed Band RNN). *A Closed Band RNN is in its architecture identical to an ordinary RNN, with the condition that the recurrent weight matrix is restricted to be a closed band matrix.*

Given that L and U are defined over φ we do not face another hyper-parameter. The overall number of parameters increases by φ^2 to $(2\varphi + 1)m$ in total. We store the non-zero values of L and U together in a φ-by-φ matrix $\hat{T}$. As for the band grid combination we carry out the multiplications before the additions. We multiply matrix $\hat{T}$ element-wise with a lower triangular matrix containing ones to filter out L, then multiply L with the head of the state matrix $(\hat{T} \cdot F_{lower})B_{head}$. Likewise we proceed for U and calculate $(\hat{T} \cdot F_{upper})B_{head}$. Both operations as well as the necessary storage are independent of m.

5.7. Shift Initialization

We close our model section with an initialization method for the closed band models using shift matrices:

Definition 5.5 (Shift matrix). *Suppose U and L are m-by-n binary matrices: $U_{ij}, L_{ij} \in 0, 1 \forall i, j \in 1, ..., mx1, ..., n$ and $k \in 1, ..., min(m, n) - 1$. Then U is an upper shift matrix of degree k and L a lower shift matrix of degree k if the following statement holds for all components:*

$$\begin{aligned} U_{ij} &= \delta_{i+k,j} \\ L_{ij} &= \delta_{i,j+k} \end{aligned} \tag{5.3}$$

U has ones only on the k-th superdiagonal and L only on the k-th subdiagonal, and both have zeroes elsewhere. They are clearly rank deficient $rank(U) = rank(L) = m - k$ containing k columns or rows containing only zeros and have a determinant of zero. Multiplied to a vector, the columns and rows filled with zeros filter out the vector head or respectively the vector tail. The rest of the vector then gets shifted up or down in the vector dimensions. That way every multiplication deletes more and more information of the vector, reducing its sub-space dimension by k every time. This also holds if multiplied with other matrices. Multiplied from the left it shifts all matrix values up or down, multiplied by right it shifts them to the left or right. Shift matrices therefore belong to a bigger group called nilpotent matrices, which when multiplied with themselves often enough turn to zero. Let's say we add new input values to the head of the hidden state. Then after $ceil(m/k)$ multiplications those values would be deleted from the hidden state, which would limit the memory capacity of a model initialized that way. We therefore combine two shift matrices, one that shifts the values of the respective units down, removing the values in the tail, and another one which shifts the values of the tail back to the head units.

Definition 5.6 (Closed Shift matrix)**.** *Suppose U and L are m-by-m shift matrices of degree $m - k$ respectively k with $k \in 1, .., m$. Then the sum of both matrices $S = U + L$, is called closed shift matrix of degree k*

Instead of being a subgroup of nilpotent matrices as before, closed shift matrices are a subgroup of permutation matrices. For each possible permutation of a vector exists exactly one permutation matrix. As they never delete or change but only rotate values, they clearly are bijective, have full rank, determinant and all eigenvalues of 1 and are therefore orthogonal. They can be multiplied infinitely to a vector and will only rotate its dimensions. If we initialize our recurrent weight matrix that way, older input information gets overwritten by new input information, but never deleted. Besides that, closed shift matrices are also a subgroup of closed band matrices and therefore a suitable initialization. We further define an input shift matrix before finally introducing the shift initialization:

Definition 5.7 (Input Shift matrix)**.** *Suppose A is an m-by-n matrix with $n < m$. We call A an input shift matrix if the following statement holds for all components:*

$$A_{ij} = \delta_{ij} \tag{5.4}$$

Definition 5.8 (Shift initialization)**.** *Suppose A is an m-by-m closed shift matrix of degree φ and B an m-by-n_i matrix with $n_i < m$ and $B_{ij} = \delta_{ij} \forall i, j \in 1, ..., mx1, ..., n_i$. If an RNN cell architecture is initialized with $W_h = A$ and $W_i = B$ we call it shift initialization.*

Figure 5.2 illustrates how the shift mechanism works. For closed band matrices the degree of the closed shift matrix is limited by the band-half-width $k < \varphi$. In practice we can set the band-half-width to match the input dimension: $\varphi = n_i = k$. An RNN cell with this initialization will be able to store $floor(m/k)$ steps completely into the

hidden state, before overwriting it with new inputs. Given this feature, it becomes clear that a higher hidden state dimension m becomes beneficial, since it increases the storage of previous inputs. Consequently this initialization matches perfectly with the closed band model properties with the closed shift matrices being a subgroup of closed band matrices and the fact that the sparsity of the model allows to increase the number of hidden units to large values while staying computationally tractable. The shift initialization could also be beneficial for classical models, with the sole drawback that they become intractable way faster as we increase the number of hidden units. We especially achieved good results with this model if we pre-trained all other parameters while holding the recurrent and input shift matrix constant. As soon as the training curves stabilized we fully enabled the training of all closed band parameters and all parameters of the input matrix. After the full storage capacity is reached, the model starts to overwrite old with new values. Therefore finding a fitting cyclicity could also increase the performance.

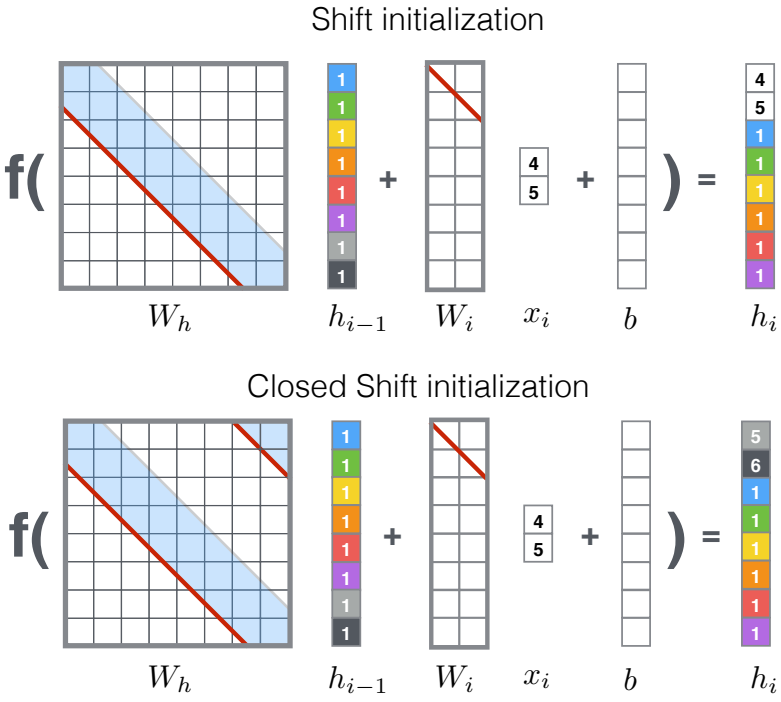

Figure 5.2.: Schematic view of the shift initialization.

6. Implementation

Band matrices and its operations are sparse, meaning that many values are restricted to be zero and many operations therefore dispensable. The standard linear algebra packages BLAS/CUBLAS do not contain a kernel optimized for the case of band-matrix matrix multiplication. Moreover, most packages do not offer a sparse format to store the band. Without efficiently using the sparsity of the band-matrix, our models will fail to stay tractable for higher dimensions. As a consequence, the implementation of a sparse storage format and of an efficient matrix multiplication kernel was mandatory. The kernels for the matrix multiplication were written in C CUDA and then exposed as dynamic-link library (DLL) to finally use them in the deep learning framework DeepNet [Urb16]. The band matrix kernels are almost completely based on the open source SGEMM kernel of the MAGMA project [NS10]. This kernel was embedded into our project and then optimized to work with band matrices. We first discuss our storage scheme, then thoroughly introduce the MAGMA SGEMM kernel. We close with the band-matrix matrix multiplication SGBMM and the matrix to band matrix multiplication SG2BMM.

6.1. Storage Schemes

We had to implement two different storage schemes. One to fully embrace the sparsity of the band matrix, containing only inside-band values. And another to work with the MAGMA kernel based foundation, which processes the matrices block-wise.

6.1.1. Sparse

Deep learning frameworks need to calculate and store the gradient with respect to all parameters. It is therefore desirable to have a format of our band matrix in which every stored value is in fact needed during calculation and all out-of-band values are removed. A blocked fashion reduces most of the zeros, but still carries many wasted values, especially in the case of small band-half-widths. We therefore implemented a kernel which takes the matrix in a completely sparse format and loads it into blocks directly before the multiplication starts. Suppose we have to store an m-by-m band matrix with band-half-width φ, which requires us to store $(2\varphi + 1)m - \varphi^2$ parameters. We then allocate those parameters in memory with single precision, requiring 4 bytes per parameter. Starting at the base pointer we can jump $4(n-1)$ bytes to reside in front of the n-th parameter. We now omit the 4 bytes factor for the jumps and just use the

indices $n \in 1, ..., (2\varphi + 1)m - \varphi^2$. Those indices then correspond to the matrix positions shown in Figure 6.1. This storage format has some beneficial properties:

- The first $n < m$ values in storage are the diagonal values, simplifying the initialization of diagonal matrices.
- For indices $n \geq m$ the upper diagonal values always have even index in storage, lower diagonal values always odd. This simplifies transposing the matrix, since only the parity for lower and upper diagonal has to be switched.
- Reusing old band matrix values in models with differing band-half-width is simplified as well. Values further away from the diagonal receive higher indices. We can therefore simply concatenate zeros to the tail for bigger or cut off the tail for lower band-half-width models.

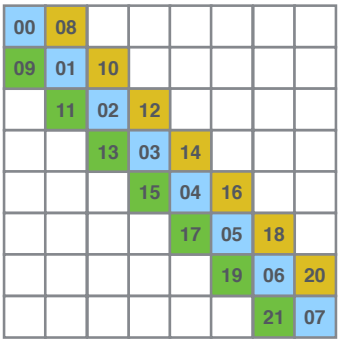

Figure 6.1.: The sparse matrix storage scheme. It shows which matrix positions correspond to which storage index.

6.1.2. Blocked band format

Matrix multiplication kernels running on GPUs divide the matrices into blocks. The workload to calculate the blocks of the resulting matrix is distributed across streaming multiprocessors (see 6.4). The matrices of the product are also processed block by block (see 6.5), to fully utilize shared memory. When a block is loaded from global to shared memory it is important that the data is coalesced, meaning that they lie next to each other in global memory. But the sparse format does not satisfy this necessary alignment condition, having values next to each other in memory which belong to opposite ends of the band matrix. Hence, we need to load the values from the sparse format into a blocked band format, satisfying the alignment condition of the magma SGEMM kernel. The block sizes of the SGEMM kernel are oriented on the warp size of 32, since those threads work most effectively together. Hence we transferred the values out of the sparse format into blocks of size 32-by-32, setting all out-of-band values in a block to zero. Figure 6.2 shows a band-matrix in full and in blocked band format (schematically

this is also shown in Figure 6.7). Values are now accessed in a row or column-wise order as typical for ordinary storage formats. Completely empty blocks are still omitted. The width of the blocked band format can be calculated as follows: $(2ceil(\varphi/32)+1)32$. As mentioned before, especially low band-half-widths lead to the highest waste of performance. With $\varphi = 1$ we nevertheless need a width of 96 for the blocked matrix, using 32 times as much values as in the sparse case.

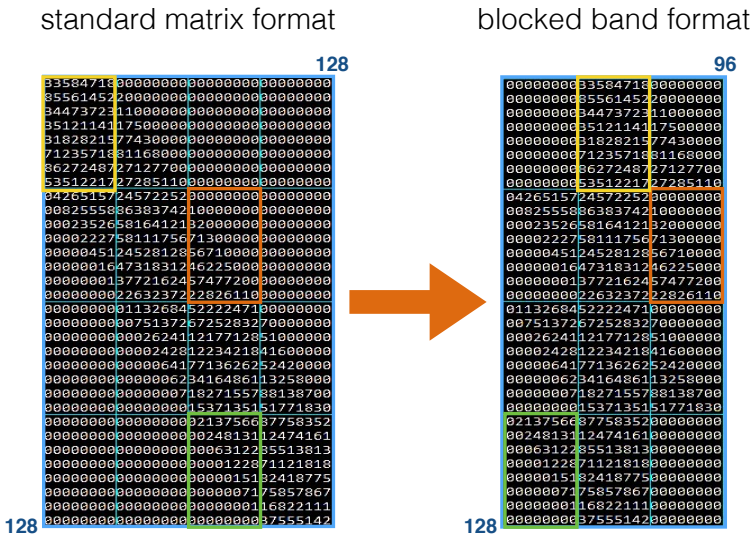

Figure 6.2.: Comparison of the band block format and the full matrix. Matrix height is 128 and band-half-width is 32. Only every fourth value is printed.

6.2. Magma SGEMM kernel

All our kernels are based on the open source magma SGEMM kernel [NS10]. Attempts to base our kernel on the MaxAs project [Gra16], an assembler to write code on register level and compile it into a CUDA binary, were omitted due to the difficult exception handling and debugging. Achieving the highest performance in 2012 the kernel is not state-of-the-art anymore. Multiplication of two 4096 dimensional squared matrices took around 109ms for the CUBLAS and 130ms for the magma kernel. GPU kernels have to be maintained to always utilize the given microarchitecture. Nevertheless, basing our work on the magma SGEMM kernel we achieved good and robust performance as can be seen in 7.2.1. To understand how this optimization works we will continue to present the basic process of the MAGMA SGEMM kernel. W.l.o.g. we suppose to multiply an m-by-m squared matrix A with an m-by-n matrix B resulting in an m-by-n matrix $C = AB$. The presented sizes for all blocks match with those we actually used. Anticipating eventual problems with similar denomination Figure 6.3 contrasts task and GPU specific entities.

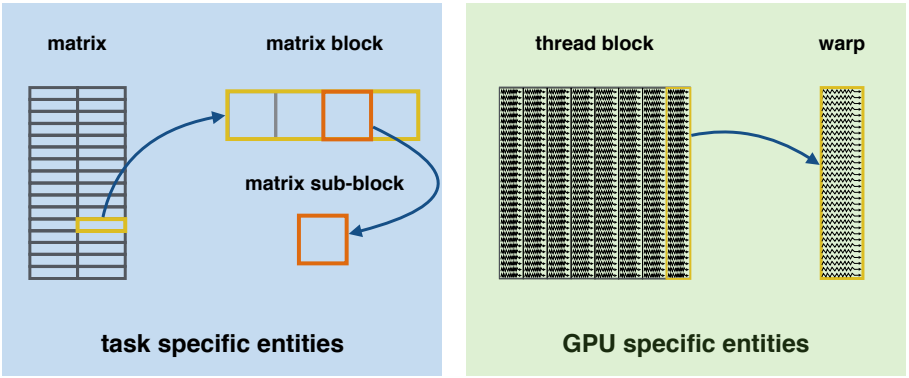

Figure 6.3.: Comparison of task and GPU specific entities.

6.2.1. Split of work

As a first step to achieve parallelism, the m-by-n matrix C is portioned into 32-by-128 blocks, as shown in Figure 6.4. The work of each block is then submitted to a streaming multiprocessor (SM). The maximum number of resident blocks and threads respectively differs for each microarchitecture. However, since the warp size was kept constant, it was always a multiple of 32. We chose to have 256 threads (8 warps) per block, e.g. leading to 8 resident blocks for a GPU capable of handling 2048 resident threads. Comparing the number of work, $32 \cdot 128$ per matrix block, with the number of threads, 256 per thread block, we find that each thread has to calculate 16 values in C. We therefore split the matrix block further down into $2 \cdot 8$ 16-by-16 sub-blocks. In Figure 6.4 we can see that neighboring values in each sub-block are calculated by neighboring threads.

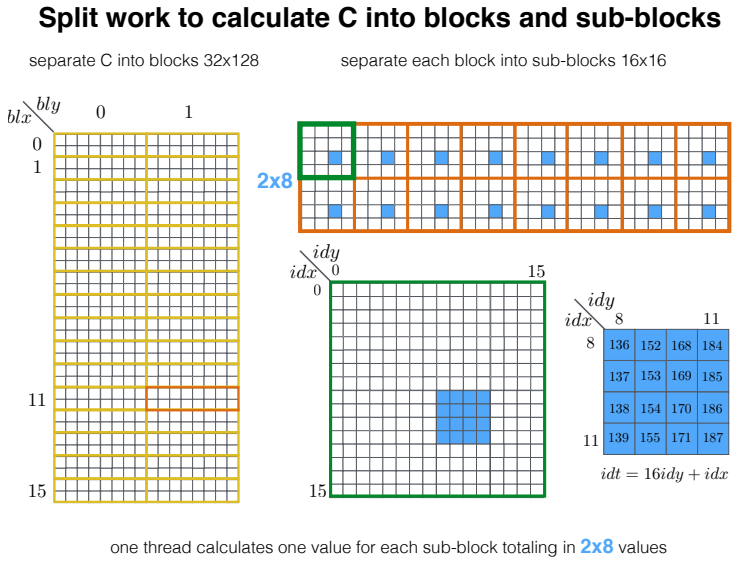

Figure 6.4.: Example to split up the work to calculate a 512-by-128 matrix. idy and idx are the thread indices and blx and bly the block indices.

6.2.2. Efficient loads

To calculate an 32-by-128 block in C we need an 32-by-m strip of A as well as an m-by-128 strip of B. For high values of m and n only small parts of those stripes fit into the fast shared memory. We therefore break down the summation of the matrix multiplication into parts, 8 summations at a time $\sum_{k=i}^{i+8}$. We load the sub-blocks of the required A and B stripes form global into shared memory and subsequently load them from shared memory to the registers to calculate the 8 summations. Warps for which the sub-block was already loaded to shared memory can calculate the multiplication, while concurrently other warps load the sub-blocks they need into shared memory. The resident warps on the GPU hide the latency of the global memory loads and keep the CUDA cores busy (background in section 4). Figure 6.5 shows how the work to load the A and B sub-blocks into shared memory is divided under the threads. Important to mention is that one warp can only efficiently load from global memory if the data is coalesced, meaning that neighboring threads have to load neighboring values in global memory. If this conditions is violated the latency grows dramatically 4.1.3, which is the exact reason why the sparse band matrix has to be pre-loaded into a blocked band format.

6.2.3. The SGEMM loop

The CUDA kernel exactly describes what each thread has to do. An overview over the different steps of the MAGMA SGEMM kernel is schematically shown in Figure 6.6. MAGMA kernels specify exactly when to transfer the matrix sub-blocks between the

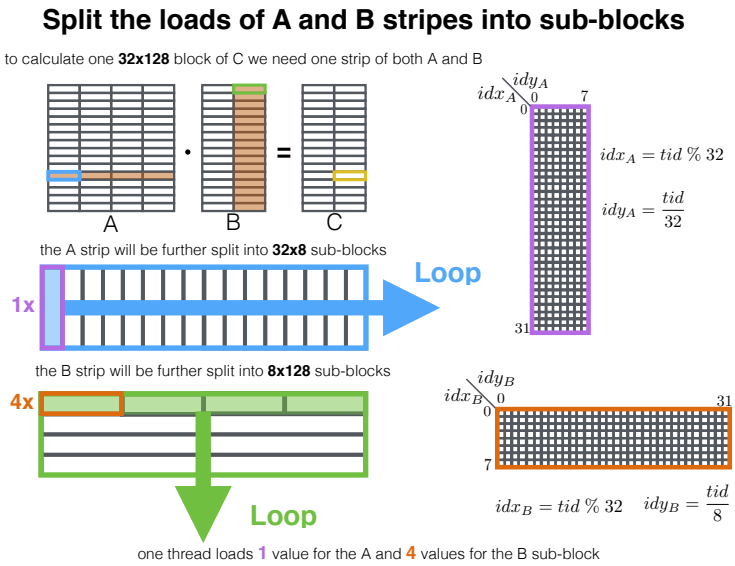

Figure 6.5.: Schematic illustration of how the stripes of A and B are loaded. The thread id tid lies between 0 and 255

different memory levels, to optimize latency hiding[NS10]. Each thread calculates 16 values of C and begins by allocating them in the registers and setting them to zero. Each thread continues to pre-load (coalesced) its share of the first sub-block of A respectively B from global into shared memory. Afterwards, the sub-block loop begins, and each thread already loads its share for the next sub-blocks into the registers, to hide the time they still have to wait for the first sub-blocks to be complete. To make sure that all threads already loaded their share of the first sub-blocks into shared memory, a synchronization barrier across all warps is set before the calculation loop starts. When all threads of the block finished to load their share, the calculation loop can start. The first 8 addends for the matrix multiplication of the 16 matrix values of C are calculated the completely loaded sub-blocks in shared memory. Subsequently each thread transfers its share of the second sub-block of A respectively B from the registers into shared memory and the loop starts from the start. While some warps occupy the CUDA cores for calculations, others already start to load their share of the next sub-blocks into shared memory. That way the waiting time of the loads is hidden behind the waiting time to carry out the calculations. The more similar the different waiting times are, the better the latency is hidden. Finding this balance is crucial and without the work of the MAGMA project our kernels would lack those properties. Finally after the sub-block loop, each thread executes a final calculation-loop and finishes by storing its sixteen values of C in global memory.

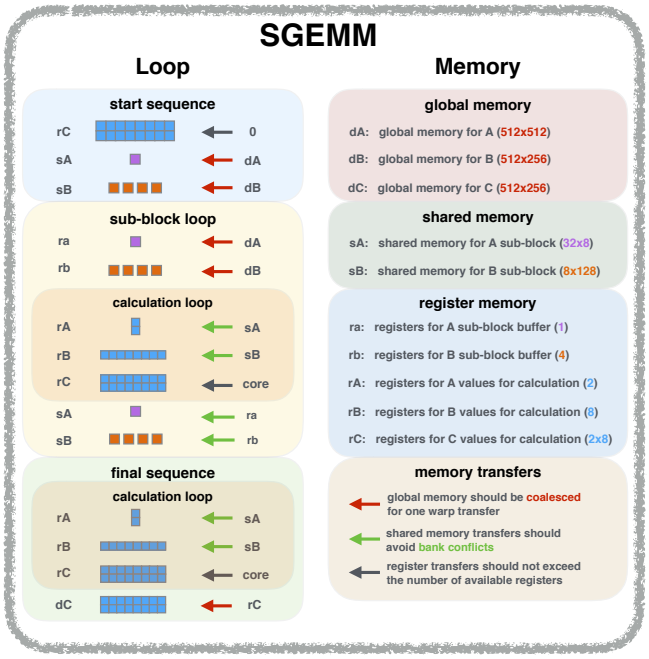

Figure 6.6.: The whole loop of the SGEMM algorithm.

6.3. Band matrix multiplication kernels

The SGBMM and the SG2BMM kernel undertake different optimizations. SGBMM shortens the sub-block loop, treating only sub-blocks of A with at least one inside-band value. SG2BMM restricts the grid of the resulting matrix C to blocks actually containing inside-band values.

6.3.1. SGBMM

Single General Band Matrix Multiplication (SGBMM) kernels receive the m-by-m band-matrix in sparse band format A_{sparse}. Before the matrix multiplication starts, is A_{sparse} transposed if selected and sequentially loaded into blocked band format $A_{sparse} \rightarrow A$. The matrix height will now be m, as in the full matrix case, but the width will decrease to $k = (2ceil(\varphi/32) + 1)32$. SGEMM loads stripes of A sized 32-by-m, which reduces in our case to 32-by-k. The length of SGEMM's sub-block loop decreases from $ceil(m/8) - 1$ to $ceil(k/8) - 1$ steps. As a consequence, we have to correct the indices of the first sub-block of A and B to load. This idea is depicted in Figure 6.7. Note that the 32-by-128 sub-block structure of B has to stay intact to avoid warp divergence.

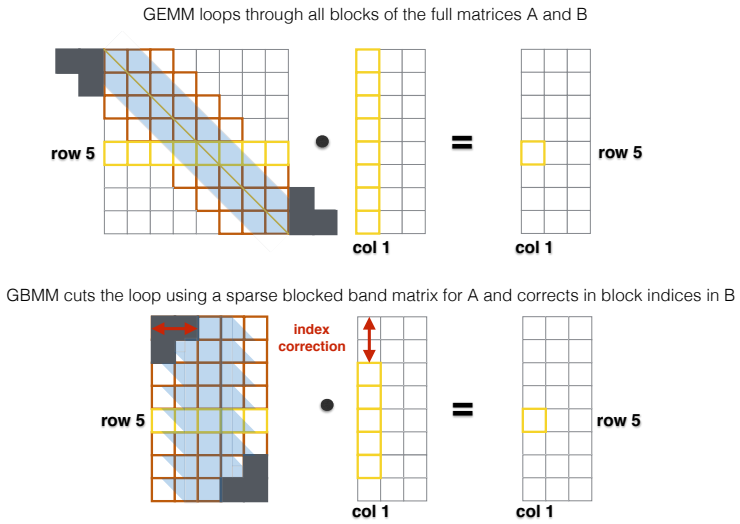

Figure 6.7.: Using a blocked band format for the band matrix and index correction we can decrease the length of the sub-block loop.

6.3.2. SG2BMM

Single General To Band Matrix Multiplication (SGBMM) kernels receive two full matrices of shape m-by-n and n-by-m. Stripes of A and B have to be loaded completely, but now

we do not require to calculate all blocks of C anymore, only those containing inside band values. The indices of C, stored in blocked band format, will not match the indices of the loaded stripes of A and B, depicted in Figure 6.8. We therefore have to alter the indices in the kernel, so that the the right block of C is calculated. After all blocks of C are computed, the blocked format is transformed back to the sparse band format $C \rightarrow C_{sparse}$. If selected, is C_{sparse} transposed.

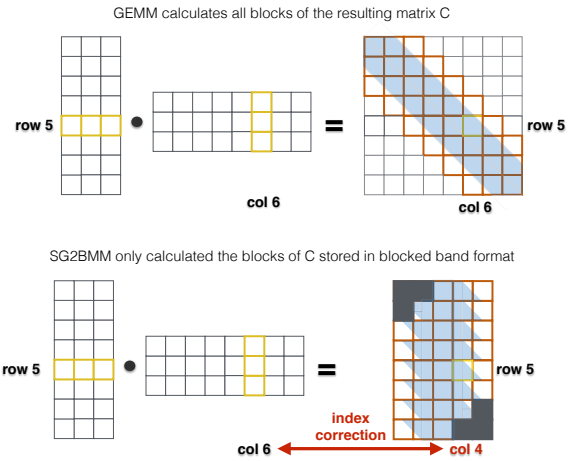

Figure 6.8.: Using a blocked band format for the band matrix and index correction we reduce the submitted blocks.

7. Experiments

After discussing the comparability of the used models we present the results of the following four experiments, each focusing on different issues:

- **Kernel benchmark:** First we checked the tractability of our implemented SGBMM and SG2BMM kernels and compared their computation time to calculate the multiplication of two m-by-m squared matrices for different sizes m to the state-of-the-art CUBLAS SGEMM kernel.
- **Model benchmark:** Next we checked how the usage of our kernels influences the overall tractability of an RNN model. We therefore executed the first training epoch of the Braille task for different numbers of hidden units and compared the by each model achieved duration.
- **Copying task:** We then tested our models on an artificial standard task for RNNs, namely the Copying task. We want to use the fact that the fixed shift initialization is a trivial solution to this task, to argue that it does not pressure models to develop interesting memory structures to solve it.
- **Braille task:** The Braille task requires end-to-end sequence learning with distorted input information and (consequentially) unaligned labels. Moreover, it contains natural data, which was measured by tactile sensors capturing 3D printed braille characters, and offers natural and artificial ways to increase its difficulty. We hope to show that the Braille task pressures the models to develop long-term memory structure and at the same time find models which develop them.

7.1. Model Comparability

Before presenting our results we want to discuss model comparability. To minimize technicalities during the presentation of our results, most details about the training algorithm and model hyper-parameters are available in the appendix A about reproducibility. All models were embedded into the exact same training environment. Nevertheless, due to their varying architectures they differ in hyper-parameters and trainable parameters.

7.1.1. Number of trainable Parameters

As suggested in other publications[CJ17] we compared our models with a fixed number of trainable parameters. Using the model formulas in 3.2, we can derive this number for

each model by summing up the parameters over all weight matrices and bias vectors. Using the abbreviations $n_{cell} = n_h^2 + n_h n_i + n_h$ and $n_{outLayer} = n_h n_o + n_o$ we find:

$$\begin{aligned} n_{RNN} &= 1 \cdot n_{cell} + n_{outLayer} \\ n_{GRU} &= 3 \cdot n_{cell} + n_{outLayer} \\ n_{LSTM} &= 4 \cdot n_{cell} + n_{outLayer} \\ n_{factorized} &= (2 \cdot n_h n_f + n_h n_i + n_h) + n_{outLayer} \\ n_{band} &= (n_h(2 \cdot \varphi + 1) - \varphi^2 + n_h n_i + n_h) + n_{outLayer} \\ n_{combi} &= n_{band} + \lambda^2 \\ n_{closed} &= n_{band} + \varphi^2 \end{aligned} \tag{7.1}$$

For all of our experiments we fixed the number of trainable parameters at $80k$. In addition with fixed input and output dimensions (n_i and n_o) and fixed other hyper-parameters (n_f, φ and λ) the size of the hidden dimension is determined for each model. Figures 7.1 and 7.2 illustrate the trends for the number of trainable parameters and the number of hidden units respectively depending on each other. Both plots are log scaled, n_f, φ and λ set to 32, input and output dimensions of 23 respectively 3 as in the Braille task.

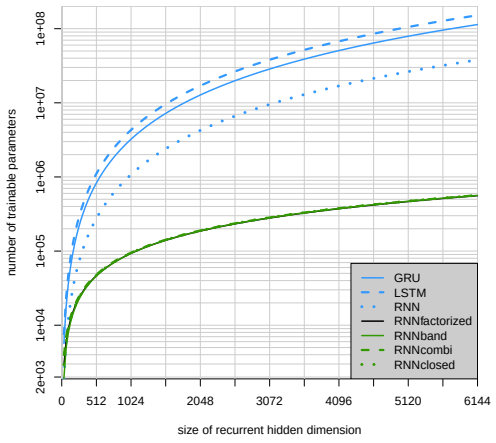

Figure 7.1.: trainable parameter trend

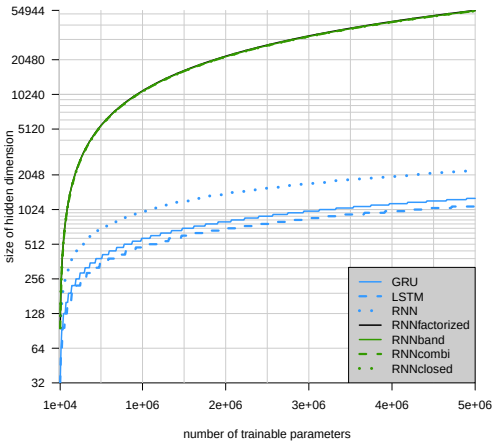

Figure 7.2.: hidden dimension trend

Remarkably, an LSTM model with 1024 hidden units has the same amount of parameters as a closed band model with close to $55k$ hidden units. During the Braille experiments we also tested closed band models with a hidden dimension of 5888, corresponding to $540k$ trainable parameters. An increase in complexity usually bears an advantage for the model. However, for almost all experiments of the Braille task most models already suffered overfitting at $80k$ trainable parameters. In this case a further increase of complexity should even worsen the test performance. Nevertheless we increased the parameters for the closed band architecture and found that the performance

on almost all tasks actually increased. The finally applied number of hidden units for each model can be found in the appendix A.

7.1.2. Hyper-Parameters

The standard RNN offers the fewest hyper-parameters. Different initialization types and trainability for each matrix and bias as well as the recurrent transfer function can be changed. All other models inherit those options. The factorized model adds one hyper-parameter for the factoring dimension n_f, the closed and open band matrix models the band-half-width φ. The combination RNN additionally has a hyper-parameter for the grid dimension λ. All three of them were fixed to 32 in all but one case, with the exception being the big closed model having used a band-half-width of 23. GRUs and LSTMs extend the RNN hyper-parameters even further, offering changes for each gate on its own and more profound changes in the gated structure itself. The detailed description of all chosen hyper-parameters for each model can be found in the appendix A.

7.1.3. Test Error

Data sets were split into *training*, *validation* and *test* set. To compare full training curves, we trained the models until the maximum epoch of 300 was reached or the training loss didn't further decrease. We then selected the configuration achieving the lowest validation loss and finally calculated the error of the model on the test set. Consequentially the reported test errors are independent of the model selection.

7.2. Benchmarks

7.2.1. Kernel Benchmark

Setting

In Figure 7.3 we compare the kernel computation times required to multiply two squared m-by-m matrices: $C = AB$. All kernels were tested for increasing matrix sizes m and executed on the same graphics card, namely an NVIDIA Quadro K2200 in TCC mode. In case of the SGBMM kernel, A was a band-matrix, in case of the SG2BMM kernel, C was a band-matrix. Unable to take the band-matrix property into account, the standard SGEMM kernel of CUBLAS needed the exact same time in both cases. We calculated results with the band-half-width kept constant at 32 and variable at $0.16 \cdot m$. The curves labeled *theoretical* are derived by applying the theoretical performance cut on the measured SGEMM performance: $t_{SGEMM} \cdot (2\varphi + 1)/m$.
In case of a product of two squared matrices, band-matrix based kernels cut the computational costs from $O(m^3)$ to $O(m^2)$. Testing the the more general case with an independent width of B would complicate the comparison by adding another parameter.

Also since the resulting matrix C was squared as well, the SGBMM and SG2BMM kernel could be compared at the same time.

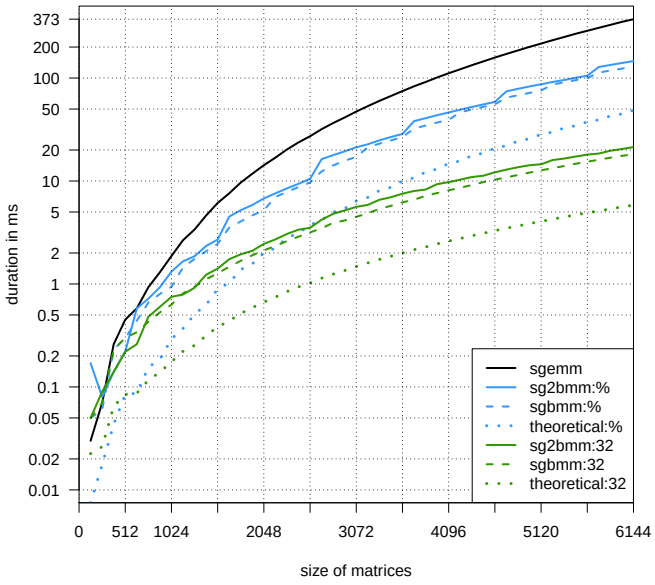

Figure 7.3.: Comparison of the kernel computation times for various matrix sizes on an NVIDIA Quadro K2200.

Discussion

For matrix sizes under 256 the state-of-the-art CUBLAS SGEMM kernel was faster than our band-matrix kernels. But after 384 the band-matrix kernels were able to outperform the CUBLAS kernel. SGBMM with a band-half-width of 32 and a size of 6144 is already around 20 times faster than SGEMM. Moreover, the theoretical performance shows that further improvements of the kernels could decrease this computational cost by another 66%.

7.2.2. Model Benchmarks

Setting

Due to the frequent occurrence of matrix multiplication in Deep Learning algorithms we expect computational speed gains in the overall models. Further operational benefits arise given that band-matrices drastically reduce the overall number of parameters and consequentially the dimension of the backpropagation gradient. Instead of growing squared $O(m^2)$ the gradient-vector will grow linearly in the number of hidden units $O(m)$. This saves storage and time during optimization. To capture this additional gain

we compare the duration of one training epoch on the Braille data set for each model. The results are shown in Figure 7.4. All models were again tested on a NVIDIA Quadro K2200 in TCC mode and the band-half-width was constant at 32.

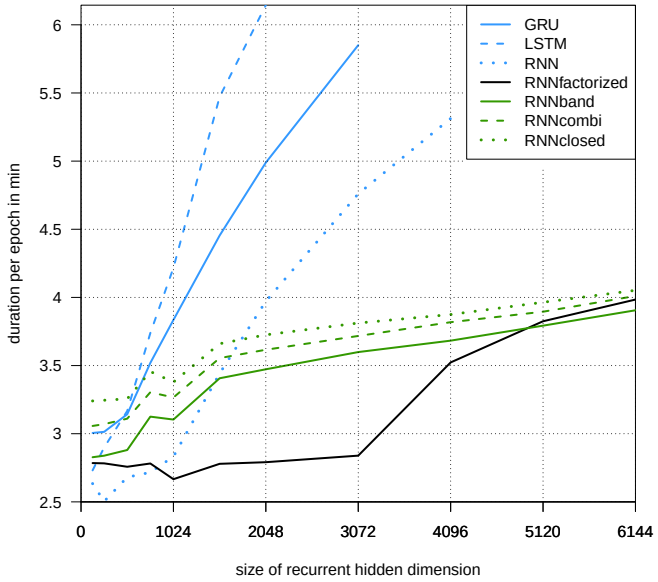

Figure 7.4.: The duration of one epoch in min for every model and different recurrent hidden sizes. Note that the missing values were caused by shortage of GPU memory.

Discussion

Classical models quickly maxed out the 4GB memory the graphics card offered: LSTMs at 2048, GRUs at 3072 and RNNs at 4096. Comparing the trends we find band-matrix models to be much more tractable. While tripling the recurrent hidden dimension from 2048 to 6144, the band-matrix models face increased computation times by less than half a minute, accounting only to an increase of less then 12.5% in overall time. On the contrary for LSTM, GRU and RNN, doubling the hidden dimension from 1024 to 2048 leads to an increase of 50%, 30% and 50% in computation time respectively. In this experiment the increase of required storage turned out to be even more problematic than the processing times themselves. We therefore couldn't explore higher scales on which we expect to see a quadratic increase of computational costs.

7.3. Copying Task

Setting

The Copying task was first proposed in [HS97]. It aims to test the models ability to store information over many steps. The first S inputs contain categorical values, chosen out of $K-2$ categories. All other $T-S$ input values are set to the same filler category $K-1$, the only exception being at step $T-S$, which is set to category K and marks the step after which the output should be given. Only the last S steps are taken as output and mirror the input sequence in the first S steps. Experiments were taken for $S = 10$, $K = 10$ and flexible overall length T. An example input and output for $T = 30$ is shown in Figure 7.5. We generated $20k$ random samples of this time for training and $2k$ each for validation and testing. Further settings concerning the models and the used training algorithm can be found in the chapter reproducibility A. Note that the effective delay between input and output is $T - S$.

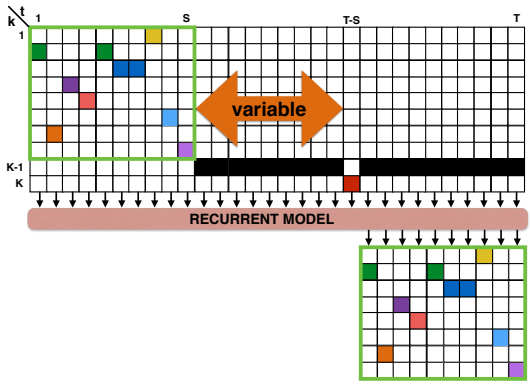

Figure 7.5.: Example input for a Copying task with $S = 10$, $K = 10$ and $T = 30$.

Discussion

Figure 7.6 shows the results of the Copying task experiment for different delays with logarithmic color scale. Due to an error, the results for all band models with trainable recurrent weight matrix are missing for the sequence lengths of 1010 and 2010. Still visible is the trend that band and combination RNN show a strong decrease in performance for delays exceeding 60 steps, for the closed band RNN for delays exceeding 210. Since it was already shown for LSTMs to perform well on the Copying task, we probably need to discuss the setting of its hyper-parameters. Despite of being more likely to fail in this task, the configuration of standard and factorize RNN should as well be investigated. The GRU on the other hand shows high performance for sequence lengths up to 2010 where it still achieves a low test error of under 3%. The closed fixed models with shift initialization are by design a solution to the copying task. While being

fixed they never forget the seen inputs and just rotate them until they have to read them out in the last S steps. We therefore argue that the Copying task does not increase the pressure to develop interesting memory structures. Note that the closed RNN with fixed shift initialization only requires 11 units and a band-half-width of 1 to completely solve this task.

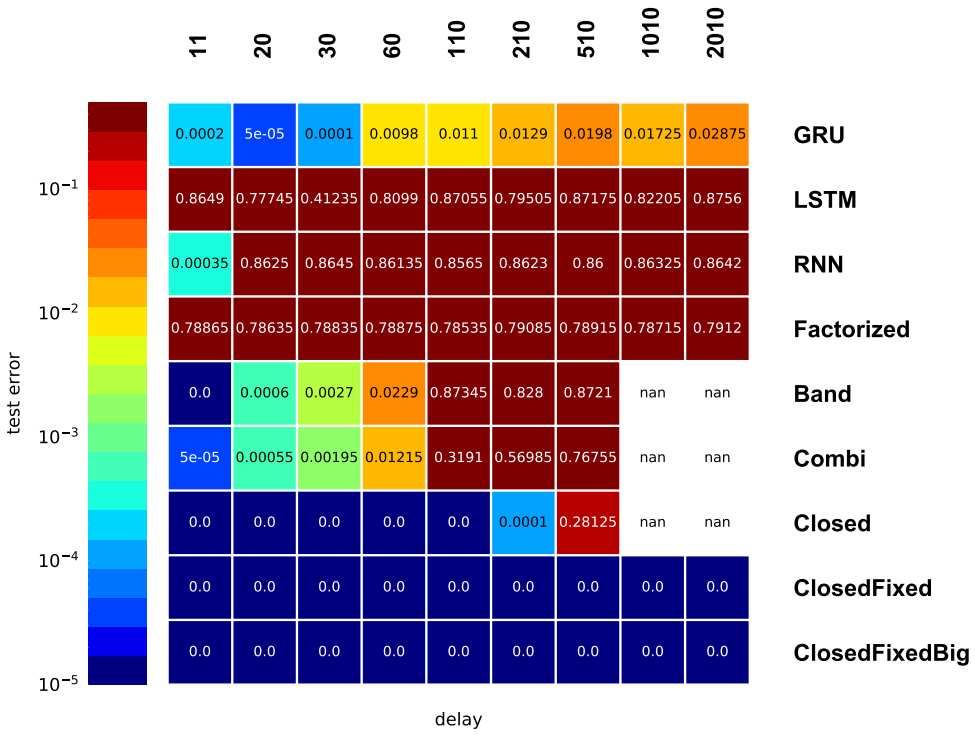

Figure 7.6.: Test errors for the Copying task. The length of delay is $T - S$.

7.4. Braille Task

7.4.1. Data Sets

Constant speed data set

The Braille data set was generated by 23 physical tactile sensors slowly sweeping over 3D printed braille characters. Over a distance of $12cm$ the sensors took 2401 equidistant values (between steps distance: $0.05mm$). Braille characters consist of three equally shaped dots, making them three dimensional binary vectors, all with a diameter of approximately $1.5mm$ (30 steps). The three dimensions are independent of each other and random, containing on average as much ones as zeros. Even though the characters are shaped like braille, they do not represent meaningful words, ruling out language correlations. We therefore look for a model which maps an 2400-by-23 dimensional continuous input onto an 2400-by-3 dimensional binary output. Given that the three dots of a braille character are independent of each other we use binary cross entropy loss:

Definition 7.1 (Binary Cross Entropy)**.** *Suppose Y is a tensor of binary output labels and $0 < \hat{Y}(X, \Theta) < 1$ an equally shaped tensor of continuous model predictions. Both tensors are shaped b-by-t-by-n, with batch size b, number of output steps t and output dimensions n . Using the sum convention we write the Binary Cross Entropy between Y and $\hat{Y}$ as:*

$$L(Y, X, \Theta) \overset{S.C.}{=} -\left(Y_{ijk} ln(\hat{Y}_{ijk}) - (1 - Y_{ijk}) ln(1 - \hat{Y_{ijk}})\right) / (b \cdot t \cdot n) \tag{7.2}$$

After training we calculate the model accuracy as follows: The 30 signals belonging to a dot are averaged and rounded, then the accuracy of the complete dots is calculated (winner takes all). Given only the first few measured steps of a braille character, the model does not have all required information to correctly predict a label. Some labels are therefore unaligned to the input and require a delayed prediction. Figure 7.7 shows an equidistant input sample after normalization, together with the respective output labels. The figure also illustrates how the delay will be applied, to make sure the model has received all input information before predicting a braille dot. The used delay posed an additional model hyper-parameter, which we included in our experiments.

Artificial Variations

The natural spread of the braille characters lead to unaligned labels. This accounts for a necessary delay of around 30 steps. To further increase the required delay we created two artificial data sets based on the Braille data set. Both transformations can be schematically seen in Figure 7.8. To apply the '**paused**' transformation we simply inserted a sequence of zeros over all input dimensions for 150 steps. This gap was placed in the middle of the equidistant Braille data set. The distortions of the second transformation were derived twofold. First we mapped the step sequence $t = 1, ..., 2401$ to distorted ones $\hat{t} = 1, f(2), ..., f(2400), 2401$ with the same start and end

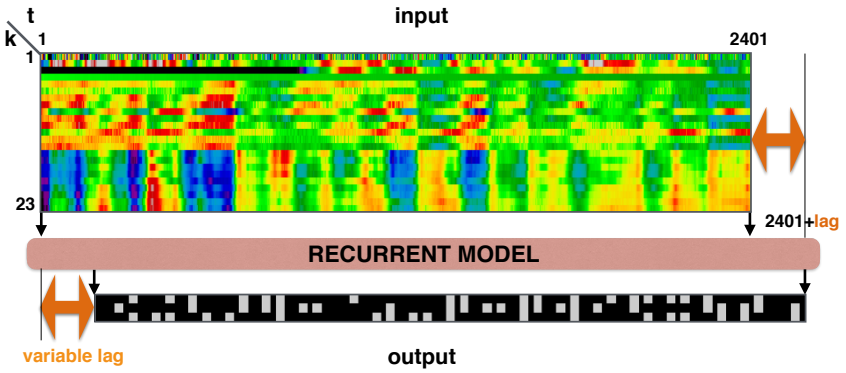

Figure 7.7.: Example input and output of the Braille data set.

points. This mapping, shown in Figure 7.8, was achieved using three linear projections, each restricted to deviate a maximum of 150 steps from the equidistant sequence. Based on the new steps we calculate the new inputs via linear interpolation:

Definition 7.2 (Linear interpolation). *Suppose the bijective function* $f : 1, ..., T \to X$ *maps each step to one input. Linear interpolation to derive* x *for a step* $t_a < t < t_b$ *with* $t_a, t_b \in 1, ..., T$ *and* $t \notin 1, ..., T$ *is given by:*

$$x = x_a + (x_b - x_a)\frac{t - t_a}{t_b - t_a} \tag{7.3}$$

An equidistant, paused and distorted Braille sample can be found in the Appendix. In combination with the model delay the pause transformation can be interpreted as a

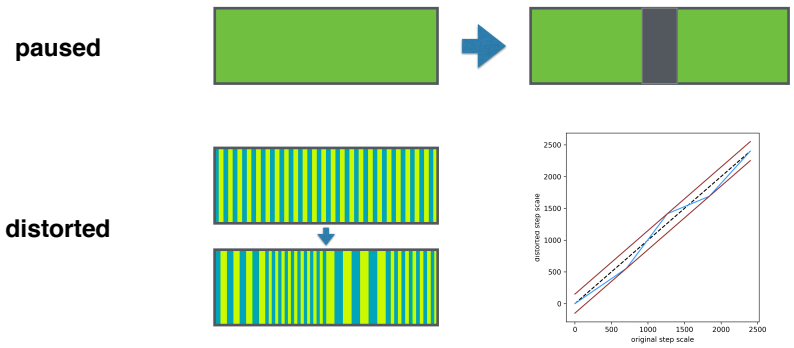

Figure 7.8.: Schematic illustration how the Braille data set input was transformed.

shift of model delay during training. Before the pause is reached, outputs are predicted 150 steps delayed to their corresponding inputs. Although the input signal is paused,

the model has to continue to be predict outputs. After the pause, the delay shrinks back to 0 and inputs have to be processed immediately. Out models are therefore required to work with two different delays at the same time: 150 steps in the first half and 0 in the second. An additional delay of 150 steps adds up to 300 in the first and 150 in the second half. The distorted case on the other hand was mapped to have the same sequence length as before. Since the inputs now also get squeezed closer together a model delay becomes mandatory to receive all information.

Variable speed data set

Inspired by the variable movement of humans we generated a more realistic Braille data set, constraining the tactile sensors to move over the surface with variable speed. While equidistant measurements suffer more noise, isochronal measurements also differ in location. The generated speed curves are unique for each sample and were restricted to begin and end the sequence at the same locations as before. Unfortunately, due to imprecision of the experimental setup, the sequence ends differ nevertheless. Figure 7.9 compares the isochronal to the equidistant location. Some curves show delays up to 2.5cm, equivalent to 500 steps. As a consequence, some curves did not capture the last 20% of the braille characters. Hence, we expect the test performances to decreased strongly. For each 3D printed pattern we took 5 for the whole data set unique speed curves. Given that reusing the underlying pattern adds cross-correlations to the samples, the training and test set were separated by the underlying pattern, making them completely independent of each other.

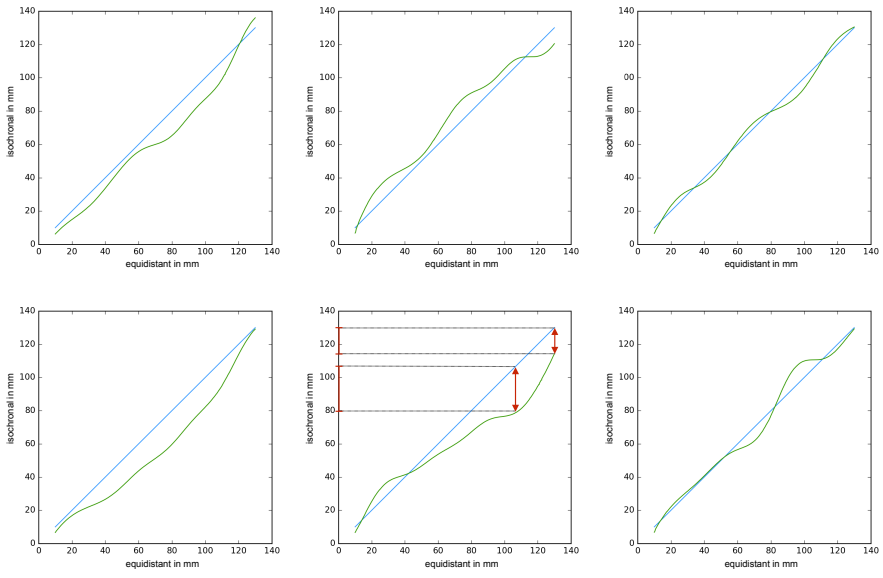

Figure 7.9.: Different curves measured by the variable speed data set. The green curve maps equidistant to isochronal, the blue shows the identity.

7.4.2. Experiments

Experiments for 9 different models were conducted on 17 different tasks. An exhaustive discussion of each case is infeasible and would go beyond the scope of this thesis. For now we want to limit ourselves to describe interesting observations in the Braille task results, moving the interpretation into the discussion chapter 8. In both cases training was carried out with normalized inputs and binary cross entropy loss. Each recurrent model used an extra sigmoidal output layer, to map the high dimensional hidden state to the three dimensional binary output. All models had a total of around $80k$ parameters, the only exception being the big closed model with $440k$ parameters. The detailed model and training configurations can be found in the appendix A. The sample size of training and test set were 56 and 14 in the constant and 280 and 50 in the variable speed data set. Given that the braille dots were completely random and independent of each other, the baseline of of the test error lies at 50%.

Observations in the Constant Braille Task

Figure 7.10 shows the test errors achieved by each model on the constant speed Braille data set. The underlying numeric values are also available as table in the appendix A. Experiments were conducted for equidistant, paused and distorted data. For the transformed data sets, the delays were increased from 100 and 200 to 150 and 300. Given that the pause causes a shift in delay only two experiments are available: shifting from 300 to 150 and from 150 to 0.

Lowest Errors:

For the equidistant data sets, the big closed continued RNN achieved the lowest error at the highest tested delay of 200 with 3.6%. The same model achieved the best result for the paused data set at a delay of 150 with a the test error of 18%. The error lays exactly 5 times higher then to the equidistant case. For artificial distortions an error of 33.5% was achieved by the small closed continued model at a delay of 150 steps. The error nearly doubles compared to the paused task, but is still substantially lower then the baseline of random guessing. Models generally decreased in performance for higher delays. This was not true for the distorted task as well as for the closed band models. The distorted task showed in general the need for a higher delay and the closed band models opposed the trend and increased in performance for higher delays in the equidistant case. The factorized model was the sole model which failed to learn any of the tasks. Except to the closed band models and the factorized RNN, all other models showed roughly the same increase respectively decrease in performance. We conclude that the tasks difficulty increased as expected and the closed band models showed the most promising results.

Observations in the Constant Braille Task

The results for the variable speed Braille data set are shown in Figure 7.11, the table again available in the appendix at A. Paused was replaced with isochronal data, the delays increased as before.

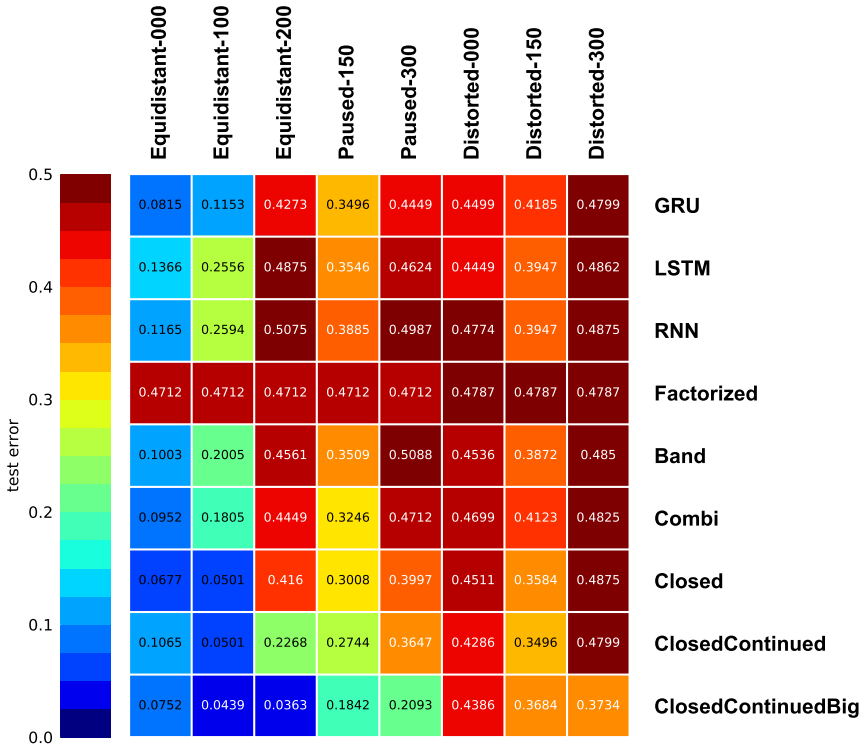

Figure 7.10.: Comparison of our major models on the Braille data set measured by a constant speed biotac sensor.

For the equidistant data sets, the big closed continued RNN achieved the lowest test error at the highest tested delay of 200 with 8.7%. The same model achieved the best result for the distorted data set at a delay of 150 with a the test error of 38.3%. The error lays around 4.5 times higher then to the equidistant case. For isochronal distortions an error of 44.0% was achieved by the small closed continued model at a delay of 200 steps. This error is already very close to the baseline error of 50%. Models generally increased in performance for median delays. This was not true for the big closed continued model in the equidistant task, the small closed continued model in the distorted task and the complete isochronal task. With one exception the factorized model failed to learn in every task. We conclude that all tasks increased in difficulty compared to constant speed braille. Especially the isochronal task was to difficult to learn.

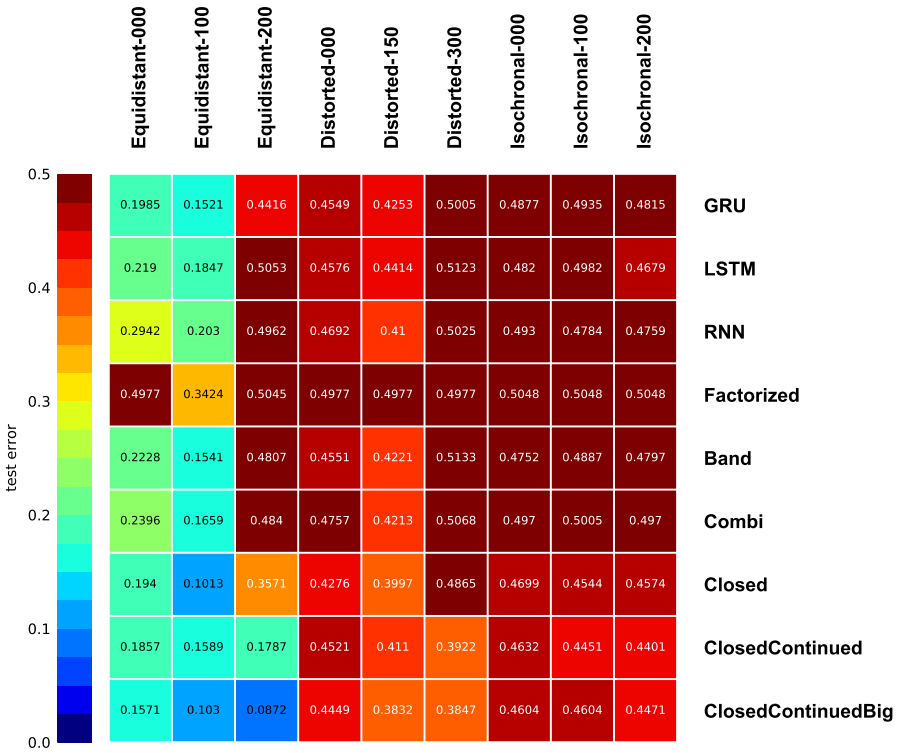

Figure 7.11.: Comparison of our major models on the Braille data set measured by a variable speed biotac sensor.

8. Discussion

8.1. Further hyper-parameter tuning

The severe non-trainability of the factorized RNN, the RNN and especially the LSTM in the Copying task, suggest that further hyper-parameter tuning is required. It was shown in [LN15] that RNNs and in [HS97] that LSTMs can perform well on the Copying task. A next step would therefore be to investigate what lead to the models deficiency. Otherwise their results in the Braille task might not be representative.
The by far the most common problem turned out to be overfitting, a well known issue, for which many solutions are proposed. One way to accomplish better generalization is to stop the training as soon as the loss on a validation set increases [Mur12]. In our case this loss increased for many models after the very first epochs or never decreased at all, making this solution infeasible.
Another way is to add regularization terms. Tests with $L1$ and $L2$ loss failed to improve performance and were removed completely, to safe hyper-parameters and additional computation time [Mur12]. Moreover, we assume that convenient regularizers won't force the model to develop better memory properties, which we expect to be the main driving force of failed generalization. Dropout is as to this date not yet implemented for the DeepNet learning environment, but could likely achieve an increase in test performance.
The previously optimization could be introduced for the classical as well as for the band models. This changes for the approach to reduce complexity. Since we fixed the number of all trainable parameters all models should show similar capability but different trainability [CJ17]. If we reduce the number of parameter, which leads to a reduction of complexity, the models are less capable to fit arbitrary patterns and forced to use simpler structures. On the other hand, the models were often not able to achieve zero training error, a sign that the model complexity is actually to low. Completely opposed to this strategy, we trained the big closed continued RNN with $540k$ instead of $80k$ parameters. But instead of suffering overfitting, this model achieved the lowest overall errors for the Braille task. A possible cause of this effect could be the shift from task to input history storage capacity, which is a basic characteristic of the band RNNs.

8.2. Band model performance

However, not all band models performed as well as the closed continued models. The ordinary band RNN as well as the grid combination RNN were outperformed by the

GRU in almost any case. This suggests that the expected deficiency of missing long term dependencies of a band RNN are not as important as expected. The increase in performance has to depend on a feature characteristic for the closed band RNN. This could be caused by the circular dependency of the hidden units or in the shift initialization mechanism, which were both exclusively used in the closed band models. Interestingly, the small closed continued and the ordinary closed model have the same dimensions, and only differ in the method, to keep the shift initialization non-trainable in the first epochs of training. We find that the closed model outperforms the closed continued model for lower delays, but the closed continued model beats it for high delays. This suggest that especially for high delays the shift initialization, in combination with the strategy of holding it fixed, leads to a higher capability to develop long-term memory. This could stem due to the fact that the fixed shift initialization effectively solves the vanishing and exploding gradient problem, which should lead to a predicted increase in trainability.
Finally, the big closed continued model offers a further comparison with the small closed continued. They both were trained with a fixed shift initialization during the first epochs, but differ drastically in the number of hidden units. We find that the big model outperforms the small model in every task in which they achieve errors lower than 34%. Furthermore, it achieves the overall lowest errors in the equidistant sets of the constant and variable Braille task, by increasing the performance once more while the delay is increased. We conclude that the high number of hidden units, and therefore the high input history storage capacity, improved the performance only while the delay was increased. The Braille task therefore obviously requires the models to develop long-term memory structure.

8.3. Summary

Our results provide evidence that the Braille task is in fact suitable to test the model capacity to develop strong memory. The best overall performance was achieved for delays of 200 steps. Moreover, at a delay of 100 steps the RNN as well as the LSTM improved on the in general harder to learn variable speed data set compared to the constant speed data set. We conclude that the variable speed data set successfully lead the models to establish a better long-term memory capacity. We suggest that varying results on rather similar tasks reveals more interesting properties. The natural and artificial distortions further expand the testing environment to a controllable increase of complexity. We argue that only tasks similar to Braille force the model to develop interesting memory structures. On this in our opinion more suitable testing model, we successfully tested band matrix models. Closed RNNs with initially fixed shift initialization and a large number of hidden units exactly meet the requirements of the Braille task. They stay tractable in the case of a large number of hidden units, due to the band matrix property. Band matrices only increase linear in computational costs with an increasing number of hidden units $O(m)$. Furthermore, they meet the requirement of

a large input history storage capacity with an only linear increase in overall trainable parameters. They inherit the property to shift the capacity from task to input history storage. Finally, they offer a simple orthogonal initialization, which solves the vanishing and exploding gradient problem and builds a long-term memory storage by design. After dropping the restriction of a shift matrix, they closed RNN still holds the band RNN properties of a close to but lower than full rank and an improved tractability and keeps the model trainable to improve the initialized storage mechanism. We could only fully utilize the tractability through our own SGBMM and SG2BMM kernels, which work with a sparse recurrent weight matrix and therefore dramatically cuts necessary storage and computations.

8.4. Outlook

So far we can't exclude the criticism that closed band models only work well on the Braille task. Hence, we should find new suitable data sets. We would thereby discourage to focus on artificial tasks, and encourage to use real world data, if necessary with artificial distortions to increase its difficulty. We already work on results for sequential MNIST, but assume that due to the one dimensional inputs the input history storage capacity might be of lower importance. Furthermore, the use in machine translation and speech recognition would be an interesting environment, but affords more special knowledge about the tasks. To improve our hitherto existing results, we should check if their performance can be further tuned due changes in hyper-parameters or training. Among others the application of dropout or other methods of regularization would be interesting for the overfitting case. Also the comparison with other models should be extended. Especially the unitary RNNs with margin could be a strong competitor of the band RNN, although they lack the advantage to use a large number of hidden units. Finally we should think about other possible extensions to our model. The approach to start at a low level of complexity and build it up step by step turned out to be very useful. Another possible extension could be to increase the band-half-width epoch by epoch, which slowly phases out the band matrix restriction.

A. Reproducibility

A.1. Configurations for the Copying Task

A.1.1. Model specific

	GRU	LSTM	RNN	Factorized	Band	Combination	Closed	Closed Cont	Big Closed Cont
Dimensionalities									
Input Features	10	10	10	10	10	10	10	10	10
Output Features	8	8	8	8	8	8	8	8	8
Input Steps	task specific	task specific	task specific	task specific	task specific	task specific	task specific	task specific	task specific
Output Steps	10	10	10	10	10	10	10	10	10
Hidden Units	160	128	256	928	1920	1824	1920	1920	5888
Factoring Dimension				32					
Gird Dimension						57			
Band-Half-Width					10	10	10	10	10
Activation Function									
Recurrent			Tanh	Tanh	Tanh	Tanh	Tanh	ReLu	ReLu
Output	Softmax	Softmax	Softmax	Softmax	Softmax	Softmax	Softmax	Softmax	Softmax
Trainability									
Weights Input	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	SWITCHED	SWITCHED
Weights Recurrent	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	SWITCHED	SWITCHED
Weights Output	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
Bias Recurrent	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	SWITCHED	SWITCHED
Bias Output	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
Weights Factor				TRUE					
Weights Corner							TRUE	SWITCHED	SWITCHED
Initialization									
Weights Input	unifrom	unifrom	unifrom	unifrom	unifrom	unifrom	shift	unifrom	unifrom
Weights Recurrent	identity	orthonormal	identity	head identity	identity	identity	shift	diagonal	diagonal
Weights Output	unifrom	unifrom	unifrom	unifrom	unifrom	unifrom	unifrom	unifrom	unifrom
Bias Recurrent	zero	zero	zero	zero	zero	zero	zero	zero	zero
Bias Output	zero	zero	zero	zero	zero	zero	zero	zero	zero
Weights Factor				head identity					
Weights Upper Corner							diagonal	diagonal	diagonal
Weights Lower Corner							zero	diagonal	diagonal

Table A.1.: Model specific configurations for the Copying task.

A.1.2. Training specific

Option	Setting
Loss	Softmax Cross Entropy
Error	Misclassification Error
Optimizer	RMSprop
Seed	1
Early Stopping	Training
Maximum Iterations	300
Minimum Iterations	30
Learning Rates	[1e-3; 1e-4; 1e-5; 1e-6]
Batch Size	100
Termination	Iteration gain 1.25
Minimum Improvement	1E-07

Table A.2.: Training specific configurations for the Copying task.

A.2. Configurations for the Braille Task

A.2.1. Model specific

	GRU	LSTM	RNN	Factorized	Band	Combination	Closed	Closed Cont	Big Closed Cont
Dimensionalities									
Input Features	23	23	23	23	23	23	23	23	23
Output Features	3	3	3	3	3	3	3	3	3
Input Steps	2401	2401	2401	2401	2401	2401	2401	2401	2401
Output Steps	2401	2401	2401	2401	2401	2401	2401	2401	2401
Hidden Units	160	128	256	864	896	864	864	864	5888
Factoring Dimension				32					
Gird Dimension						27			
Band-Half-Width					32	32	32	32	23
Activation Function									
Recurrent			Tanh	Tanh	Tanh	Tanh	Tanh	ReLu	ReLu
Output	Sigmoid	Sigmoid	Sigmoid	Sigmoid	Sigmoid	Sigmoid	Sigmoid	Sigmoid	Sigmoid
Trainability									
Weights Input	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	SWITCHED	SWITCHED
Weights Recurrent	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	SWITCHED	SWITCHED
Weights Output	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
Bias Recurrent	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	SWITCHED	SWITCHED
Bias Output	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
Weights Factor				TRUE					
Weights Corner							TRUE	SWITCHED	SWITCHED
Initialization									
Weights Input	unifrom	unifrom	unifrom	unifrom	unifrom	unifrom	shift	unifrom	unifrom
Weights Recurrent	identity	orthonormal	identity	head identity	identity	identity	shift	diagonal	diagonal
Weights Output	unifrom	unifrom	unifrom	unifrom	unifrom	unifrom	unifrom	unifrom	unifrom
Bias Recurrent	zero	zero	zero	zero	zero	zero	zero	zero	zero
Bias Output	zero	zero	zero	zero	zero	zero	zero	zero	zero
Weights Factor				head identity					
Weights Upper Corner							diagonal	diagonal	diagonal
Weights Lower Corner							zero	diagonal	diagonal

Table A.3.: Model specific configurations for the Braille task.

A.2.2. Training specific

Option	Setting
Loss	Binary Cross Entropy
Error	Misclassification Error
Optimizer	RMSprop
Seed	1
Early Stopping	Training
Maximum Iterations	300
Minimum Iterations	30
Learning Rates	[1e-3; 1e-4; 1e-5; 1e-6]
Batch Size	7
Termination	Iteration gain 1.25
Minimum Improvement	1E-07

Table A.4.: Training specific configurations for the Braille task.

A.3. Results in tabular form

	1	10	20	50	100	200	500	1000	2000
GRU	0.02	0	0.01	0.98	1.1	1.29	1.98	1.73	2.88
LSTM	86.49	77.74	41.24	80.99	87.06	79.51	87.18	82.2	87.56
RNN	0.03	86.25	86.45	86.14	85.65	86.23	86	86.32	86.42
Factorized	78.87	78.64	78.84	78.88	78.54	79.09	78.91	78.72	79.12
Band	0	0.06	0.27	2.29	87.34	82.8	87.21	nan	nan
Combi	0	0.06	0.2	1.22	31.91	56.98	76.76	nan	nan
Closed	0	0	0	0	0	0.01	28.12	nan	nan
ClosedFixed	0	0	0	0	0	0	0	0	0
ClosedFixedBig	0	0	0	0	0	0	0	0	0

Table A.5.: Test error table of the Copying task.

	Equidistant-000	Equidistant-100	Equidistant-200	Paused-150	Paused-300	Distorted-000	Distorted-150	Distorted-300
GRU	8.15	11.53	42.73	34.96	44.49	44.99	41.85	47.99
LSTM	13.66	25.56	48.75	35.46	46.24	44.49	39.47	48.62
RNN	11.65	25.94	50.75	38.85	49.87	47.74	39.47	48.75
Factorized	47.12	47.12	47.12	47.12	47.12	47.87	47.87	47.87
Band	10.03	20.05	45.61	35.09	50.88	45.36	38.72	48.5
Combi	9.52	18.05	44.49	32.46	47.12	46.99	41.23	48.25
Closed	6.77	5.01	41.6	30.08	39.97	45.11	35.84	48.75
ClosedContinued	10.65	5.01	22.68	27.44	36.47	42.86	34.96	47.99
ClosedContinuedBig	7.52	4.39	3.63	18.42	20.93	43.86	36.84	37.34

Table A.6.: Test error table of the Constant Braille task.

	Equidistant-000	Equidistant-100	Equidistant-200	Distorted-000	Distorted-150	Distorted-300	Varspeed-000	Varspeed-100	Varspeed-200
GRU	19.85	15.21	44.16	45.49	42.53	50.05	48.77	49.35	48.15
LSTM	21.9	18.47	50.53	45.76	44.14	51.23	48.2	49.82	46.79
RNN	29.42	20.3	49.62	46.92	41	50.25	49.3	47.84	47.59
Factorized	49.77	34.24	50.45	49.77	49.77	49.77	50.48	50.48	50.48
Band	22.28	15.41	48.07	45.51	42.21	51.33	47.52	48.87	47.97
Combi	23.96	16.59	48.4	47.57	42.13	50.68	49.7	50.05	49.7
Closed	19.4	10.13	35.71	42.76	39.97	48.65	46.99	45.44	45.74
ClosedContinued	18.57	15.89	17.87	45.21	41.1	39.22	46.32	44.51	44.01
ClosedContinuedBig	15.71	10.3	8.72	44.49	38.32	38.47	46.04	46.04	44.71

Table A.7.: Test error table of the Variable Braille task.

B. Code

Since the matter of publication is undecided the code is stored in a private repository on github: `https://github.com/BRML/DeepPrivate`. We used the F# deep learning framework DeepNet [Urb16], implemented by Sebastian Urban. Further we created a private C++ project containing our CUDA kernels together with a testing framework, which creates a link-able DLL file for our kernels: `https://github.com/BRML/BandBlas`. All models were implemented from scratch using the expressions of DeepNet, which offer to build a differentiable computational graph and use backpropagation for training. The models were run on a cluster using Microsoft HPC Pack on nodes with GPUs of varying compute capabilities (from 3.0 to 5.0). To review code and the used software package versions we recommend to join the specific github projects.

List of Figures

List of Tables

Bibliography

[AA16] M. Arjovsky and Y. B. Amar Shah. *Unitary Evolution Recurrent Neural Networks*. arXiv:1511.06464 [cs.LG], 2016.

[AK16] G. Ansmann and U. F. Klaus Lehnertz. *Self-induced switchings between multiple space-time patterns on complex networks of excitable units*. Phys. Rev. X 6, 011030, 2016.

[AR15] D. Amodei and e. a. Rishita Anubhai Eric Battenberg. *Deep Speech 2: End-to-End Speech Recognition in English and Mandarin*. arXiv:1512.02595 [cs.CL], 2015.

[Bal11] J. Balfour. *Introduction to CUDA*. NVIDIA Research,CME343,ME339, 2011.

[BK15] D. Bahdanau and Y. B. Kyunghyun Cho. *Neural Machine Translation by Jointly Learning to Align and Translate*. Published as a conference paper at ICLR 2015, 2015.

[Bro14] E. D. Brown. *Technology, Strategy, People, Projects*. `http://ericbrown.com/drowning-in-data-starved-for-information.htm`, 2014.

[BSF94] Y. Bengio, P. Simard, and P. Frasconi. *Learning long-term dependencies with gradient descent is difficult*. IEEE transactions on neural networks, 5(2):157-166, 1994.

[CB14] K. Cho and e. a. Bart van Merrienboer Caglar Gulcehre. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*. arXiv:1406.1078 [cs.CL], 2014.

[CJ17] J. Collins and D. S. Jascha Sohl-Dickstein. *Capacity and Trainability in Recurrent Neural Networks*. 2017, Published as a conference paper at ICLR 2017.

[Cyb89] G. Cybenko. *Approximation by superpositions of a sigmoidal function*. Math. Control Signal Systems (1989) 2: 303., 1989.

[Die15] J. B. Diederik P. Kingma. *Adam: A Method for Stochastic Optimization*. Conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[EP66] L. E.Baum and T. Petrie. *Statistical Inference for Probabilistic Functions of Finite State Markov Chains*. The Annals of Mathematical Statistics. 37 (6): 1554–1563, 1966.

[Gra16] S. Gray. *Assembler for NVIDIA Maxwell architecture*. `https://github.com/NervanaSystems/maxas`, 2016.

[GS06] A. Graves and J. S. Santiago Fernández Faustino Gomez. *Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks*. ICML '06 Proceedings of the 23rd international conference on Machine Learning, 2006.

[HA16] M. Henaff and Y. L. Arthur Szlam. *Recurrent Orthogonal Networks and Long-Memory Tasks*. Proceedings of the 33rd International Conference on Machine Learning, JMLR, 2016.

[Hin06] T. Y. Hinton GE1 Osindero S. *A fast learning algorithm for deep belief nets.* Neural Comput. 2006 Jul;18(7):1527-54., 2006.

[HM89] K. Hornik and H. W. Maxwell Stinchcombe. *Multilayer feedforward networks are universal approximators*. Neural Networks, Volume 2, Issue 5, 1989.

[HS97] S. Hochreiter and J. Schmidhuber. *Long short-term memory*. Neural computation 9(8):1735-1780, 1997.

[HX15] K. He and J. S. Xiangyu Zhang Shaoqing Ren. *Deep Residual Learning for Image Recognition*. arXiv:1512.03385 [cs.CV], 2015.

[Ily45] e. a. Ilya Nikolaevich Bronshtein Konstantin Adolfovic Semendyayev. *Handbook of Mathematics*. 5. Aufl., B. G. Teubner Verlagsgesellschaft, 1945.

[KG17] O. Kuchaiev and B. Ginsburg. *Factorization tricks for LSTM networks*. ICLR 2017 Workshop, 2017.

[KM15] D. Krueger and R. Memisevic. *Regularizing RNNs by Stabilizing Activations*. arXiv:1511.08400 [cs.NE], 2015.

[LL98] Y. LeCun and P. H. Léon Bottou Yoshua Bengio. *Gradient-based learning applied to document recognition*. Proceedings of the IEEE, 86(11):2278-2324, November, 1998.

[LN15] Q. V. Le and G. E. H. Navdeep Jaitly. *A Simple Way to Initialize Recurrent Networks of Rectified Linear Units*. arXiv:1504.00941 [cs.NE], 2015.

[LV16] Z. Lu and T. N. S. Vikas Sindhwani. *Learning Compact Recurrent Neural Networks*. arXiv:1604.02594v1 [cs.LG], 2016.

[MR15] T. Mowry and B. Railing. *Parallel Computer Architecture and Programming (CMU 15-418/618)*. `http://15418.courses.cs.cmu.edu/fall2017/`, 2015.

[Mur12] K. P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[Nai82] J. Naisbitt. *Megatrends: Ten New Directions Transforming Our Lives*. Warner Books, Inc., 1982.

[Nis05] Y. Nishimori. *A Note on Riemannian Optimization Methods on the Stiefel and the Grassmann Manifolds*. 2005 International Symposium on Nonlinear Theory and its Applications (NOLTA2005), 2005.

[NS10] R. Nath and J. D. Stanimire Tomov. *An Improved MAGMA GEMM For Fermi Graphics Processing Units*. Int. J. High Perform. Comput. Appl., 2010.

[NVI09] NVIDIA. *NVIDIA Fermi Compute Architecture Whitepaper*. `http://www.nvidia.de/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2009.

[nVi17a] nVidia. *CUDA C Best Practices Guide*. `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html`, 2017.

[nVi17b] nVidia. *CUDA C Programming Guide*. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`, 2017.

[nVi17c] nVidia. *CUDA Toolkit Documentation v9.0.176*. `http://docs.nvidia.com/cuda/`, 2017.

[PT13] R. Pascanu and Y. B. Tomas Mikolov. *On the difficulty of training recurrent neural networks*. ICML (3), 28:1310-1318, 2013.

[RG86] D. E. Rumelhart and R. J. W. Geoffrey E. Hinton. *Learning representations by back-propagating errors*. Nature 323, 533 - 536 (09 October), 1986.

[RS16] J. Redmon and A. F. Santosh Divvala Ross Girshick. *You Only Look Once: Unified, Real-Time Object Detection*. arXiv:1506.02640 [cs.CV], 2016.

[Sch15] J. Schmidhuber. *Deep Learning in Neural Networks: An Overview*. Neural Networks Vol. 61 85-117, 2015.

[Sze14] C. Szegedy and e. a. Wei Liu Yangqing Jia. *Going Deeper with Convolutions*. arXiv:1409.4842 [cs.CV], 2014.

[Tag11] H. D. Tagare. *Notes on Optimization on Stiefel Manifolds*. Technical report, Yale University, 2011.

[TD95] S. H. T. and S. E. D. *On the Computational Power of Neural Nets*. Journal of Computer and System Sciences, Volume 50, Issue 1, February, Pages 132-150, 1995.

[TH12] T. Tieleman and G. Hinton. *Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude*. COURSERA: Neural Networks for Machine Learning, 2012.

[Urb16] S. Urban. *Deep.Net machine learning framework for F#*. `https://github.com/DeepMLNet/DeepNet`, 2016.

[VC17] E. Vorontsov and C. P. Chiheb Trabelsi Samuel Kadoury. *On orthogonality and learning recurrent networks with long term dependencies*. arXiv:1702.00071 [cs.LG], 2017.

[WT16] S. Wisdom and e. a. Thomas Powers. *Full-Capacity Unitary Recurrent Neural Networks*. 30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain, 2016.

[ZI15] W. Zaremba and O. V. Ilya Sutskever. *Recurrent neural network regularization*. arXiv:1409.2329 [cs.NE], 2015.